VISIT…

THE EXPERT'S VOICE® IN OPEN SOURCE

The Definitive Guide to GCC

Everything you need to know about using the GNU Compiler Collection and related tools

SECOND EDITION

William von Hagen

Apress®

The Definitive Guide to GCC

Second Edition

William von Hagen

Apress®

The Definitive Guide to GCC, Second Edition

Copyright © 2006 by William von Hagen

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-59059-585-5

ISBN-10 (pbk): 1-59059-585-8

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editors: Jason Gilmore, Keir Thomas
Technical Reviewer: Gene Sally
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Richard Dal Porto
Copy Edit Manager: Nicole LeClerc
Copy Editor: Jennifer Whipple
Assistant Production Director: Kari Brooks-Copony
Production Editor: Katie Stence
Compositor: Susan Glinert
Proofreader: Elizabeth Berry
Indexer: Toma Mulligan
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The information in this book is distributed on an "as is" basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at http://www.apress.com in the Source Code section.

To Dorothy Fisher, for all your love, support, and encouragement.
And for Becky Gable—what would we do without the schematics?
—Bill von Hagen

Contents at a Glance

About the Author xvii
About the Technical Reviewer xix
Acknowledgments xxi
Introduction xxiii

CHAPTER 1 Using GCC's C Compiler 1
CHAPTER 2 Using GCC's C++ Compiler 41
CHAPTER 3 Using GCC's Fortran Compiler 53
CHAPTER 4 Using GCC's Java Compiler 79
CHAPTER 5 Optimizing Code with GCC 101
CHAPTER 6 Analyzing Code Produced with GCC Compilers 119
CHAPTER 7 Using Autoconf and Automake 151
CHAPTER 8 Using Libtool 177
CHAPTER 9 Troubleshooting GCC 197
CHAPTER 10 Additional GCC and Related Topic Resources 215
CHAPTER 11 Compiling GCC 227
CHAPTER 12 Building and Installing Glibc 247
CHAPTER 13 Using Alternate C Libraries 281
CHAPTER 14 Building and Using C Cross-Compilers 299
APPENDIX A Using GCC Compilers 321
APPENDIX B Machine- and Processor-Specific Options for GCC 403
APPENDIX C Using GCC's Online Help 491

INDEX 505

Contents

About the Author xvii
About the Technical Reviewer xix
Acknowledgments xxi
Introduction xxiii

CHAPTER 1 Using GCC's C Compiler 1

GCC Option Refresher 1
Compiling C Dialects 3
Exploring C Warning Messages 7
GCC's C and Extensions 10
Locally Declared Labels 11
Labels As Values 12
Nested Functions 13
Constructing Function Calls 14
Referring to a Type with typeof 15
Zero-Length Arrays 15
Arrays of Variable Length 17
Macros with a Variable Number of Arguments 18
Subscripting Non-Ivalue Arrays 18
Arithmetic on Void and Function Pointers 19
Nonconstant Initializers 19
Designated Initializers 19
Case Ranges 21
Mixed Declarations and Code 21
Declaring Function Attributes 21
Specifying Variable Attributes 25
Inline Functions 27
Function Names As Strings 28
#pragmas Accepted by GCC 29
Objective-C Support in GCC's C Compiler 30
Compiling Objective-C Applications 32
GCC Options for Compiling Objective-C Applications 33
Exploring the GCC Objective-C Runtime 36

CHAPTER 2 Using GCC's C++ Compiler 41

GCC Option Refresher 41
Filename Extensions for C++ Source Files 43
Command-Line Options for GCC's C++ Compiler 43
ABI Differences in g++ Versions 46
GNU C++ Implementation Details and Extensions 47
Attribute Definitions Specific to g++ 47
C++ Template Instantiation in g++ 49
Function Name Identifiers in C++ and C 49
Minimum and Maximum Value Operators 50
Using Java Exception Handling in C++ Applications 50
Visibility Attributes and Pragmas for GCC C++ Libraries 51

CHAPTER 3 Using GCC's Fortran Compiler 53

Fortran History and GCC Support 54
Compiling Fortran Applications with gfortran 55
Common Compilation Options with Other GCC Compilers 55
Sample Code 57
Compiling Fortran Code 57
Modernizing the Sample Fortran Code 59
Command-Line Options for gfortran 62
Code Generation Options 62
Debugging Options 63
Directory Search Options 63
Fortran Dialect Options 63
Warning Options 64
gfortran Intrinsics and Extensions 65
Classic GNU Fortran: The g77 Compiler 74
Why Use g77? 74
Differences Between g77 and gfortran Conventions 74
Alternatives to gfortran and g77 75
The f2c Fortran-to-C Conversion Utility 76
The g95 Fortran Compiler 76
Intel's Fortran Compiler 76
Additional Sources of Information 77

CHAPTER 4 Using GCC's Java Compiler 79

Java and GCC's Java Compiler 79
Basic gcj Compiler Usage 80
Demonstrating gcj, javac, and JVM Compatibility 83
Filename Extensions for Java Source Files 86
Command-Line Options for GCC's Java Compiler 86
Constructing the Java Classpath 89
Creating and Using Jar Files and Shared Libraries 90
GCC Java Support and Extensions 92
- Java Language Standard ABI Conformance 93
- Runtime Customization 93
- Getting Information About Java Source and Bytecode Files 94
- Using the GNU Interpreter for Java 96

Java and C++ Integration Notes 98

CHAPTER 5 Optimizing Code with GCC 101

A Whirlwind Tour of Compiler Optimization Theory 102
- Code Motion 103
- Common Subexpression Elimination 103
- Constant Folding 103
- Copy Propagation Transformations 104
- Dead Code Elimination 104
- If-Conversion 105
- Inlining 105

GCC Optimization Basics 105
What's New in GCC 4.x Optimization 106
Architecture-Independent Optimizations 106
- Level 1 GCC Optimizations 107
- Level 2 GCC Optimizations 109
- GCC Optimizations for Code Size 111
- Level 3 GCC Optimizations 112
- Manual GCC Optimization Flags 112

Processor-Specific Optimizations 113
Automating Optimization with Acovea 114
- Building Acovea 114
- Configuring and Running Acovea 115

CHAPTER 6 Analyzing Code Produced with GCC Compilers 119

Test Coverage Using GCC and gcov 120
Overview of Test Coverage 120
Compiling Code for Test Coverage Analysis 123
Using the gcov Test Coverage Tool 124
A Sample gcov Session 126
Files Used and Produced During Coverage Analysis 133
Code Profiling Using GCC and gprof 133
Obtaining and Compiling gprof 134
Compiling Code for Profile Analysis 135
Using the gprof Code Profiler 136
Symbol Specifications in gprof 136
A Sample gprof Session 140
Displaying Annotated Source Code for Your Applications 144
Adding Your Own Profiling Code Using GCC's C Compiler 148
Mapping Addresses to Function Names 148
Common Profiling Errors 149

CHAPTER 7 Using Autoconf and Automake 151

Introducing Unix Software Configuration, Autoconf, and Automake 151
Installing and Configuring autoconf and automake 154
Deciding Whether to Upgrade or Replace autoconf and automake 154
Building and Installing autoconf 155
Obtaining and Installing Automake 158
Configuring Software with autoconf and automake 161
Creating Configure.ac Files 161
Creating Makefile.am Files and Other Files Required by automake 166
Running Autoconf and Automake 169
Running Configure Scripts 174

CHAPTER 8 Using Libtool 177

Introduction to Libraries 177
Static Libraries 177
Shared Libraries 178
Dynamically Loaded Libraries 180
What Is Libtool? 181
Downloading and Installing Libtool 182
Installing Libtool 182
Files Installed by Libtool 184

Using Libtool 185
Using Libtool from the Command Line 185
Command-Line Options for Libtool 186
Command-Line Modes for Libtool Operation 186
Using Libtool with Autoconf and Automake 191
Troubleshooting Libtool Problems 194
Getting More Information About Libtool 195

CHAPTER 9 Troubleshooting GCC 197

Coping with Known Bugs and Misfeatures 198
Using -### to See What's Going On 199
Resolving Common Problems 200
Problems Executing GCC 200
Using Multiple Versions of GCC on a Single System 200
Problems Loading Libraries When Executing Programs 201
'No Such File or Directory' Errors 202
Problems Executing Files Compiled with GCC Compilers 203
Running Out of Memory When Using GCC 203
Moving GCC After Installation 204
General Issues in Mixing GNU and Other Toolchains 204
Specific Compatibility Problems in Mixing GCC with Other Tools 206
Problems When Using Optimization 208
Problems with Include Files or Libraries 208
Mysterious Warning and Error Messages 209
Incompatibilities Between GNU C and K&R C 210
Abuse of the __STDC__ Definition 211
Resolving Build and Installation Problems 212

CHAPTER 10 Additional GCC and Related Topic Resources 215

Usenet Resources for GCC 215
Selecting Software for Reading Usenet News 216
Summary of GCC Newsgroups 217
Mailing Lists for GCC 219
GCC Mailing Lists at gcc.gnu.org 219
Netiquette for the GCC Mailing Lists 222
Other GCC-Related Mailing Lists 223
World Wide Web Resources for GCC and Related Topics 223
Information About GCC and Cross-Compilation 224
Information About Alternate C Libraries 225
Publications About GCC and Related Topics 225

CHAPTER 11 Compiling GCC 227

Why Build GCC from Source? 227
Starting the Build Process 228
Verifying Software Requirements 228
Preparing the Installation System 230
Downloading the Source Code 231
Installing the Source Code 231
Configuring the Source Code 232
What Is in a (System) Name? 233
Additional Configuration Options 234
NLS-Related Configuration Options 239
Building Specific Compilers 239
Compiling the Compilers 239
Compilation Phases 240
Other Make Targets 241
Testing the Build 242
Installing GCC 245

CHAPTER 12 Building and Installing Glibc 247

What Is in Glibc? 247
Why Build Glibc from Source? 249
Potential Problems in Upgrading Glibc 250
Identifying Which Glibc a System Is Using 251
Getting More Details About Glibc Versions 252
Glibc Add-Ons 253
Previewing the Build Process 254
Recommended Tools for Building Glibc 256
Updating GNU Utilities 257
Downloading and Installing Source Code 258
Downloading the Source Code 258
Installing Source Code Archives 258
Integrating Add-Ons into the Glibc Source Code Directory 260
Configuring the Source Code 261
Compiling Glibc 264
Testing the Build 265
Installing Glibc 265
Installing Glibc As the Primary C Library 266
Installing an Alternate Glibc 268
Using a Rescue Disk 269

Troubleshooting Glibc Installation Problems 270
Resolving Upgrade Problems Using BusyBox 271
Resolving Upgrade Problems Using a Rescue Disk 273
Backing Out of an Upgrade 274
Problems Using Multiple Versions of Glibc 276
Getting More Information About Glibc 276
Glibc Documentation 277
Other Glibc Web Sites 277
Glibc Mailing Lists 277
Reporting Problems with Glibc 278
Moving to Glibc 2.4 278

CHAPTER 13 Using Alternate C Libraries 281

Why Use a Different C Library? 281
Overview of Alternate C Libraries 282
Overview of Using Alternate C Libraries 282
Building and Using dietlibc 283
Getting dietlibc 284
Building dietlibc 284
Using dietlibc with gcc 285
Building and Using klibc 286
Getting klibc 286
Building klibc 287
Using klibc with gcc 288
Building and Using Newlib 289
Getting Newlib 289
Building and Using Newlib 290
Building and Using uClibc 290
Getting uClibc 291
Building uClibc 292
Using uClibc with gcc 296

CHAPTER 14 Building and Using C Cross-Compilers 299

What Is Cross-Compilation? 299
Using crosstool to Build Cross-Compilers 300
Retrieving the crosstool Package 304
Building a Default Cross-Compiler Using crosstool 304
Building a Custom Cross-Compiler Using crosstool 305
Using buildroot to Build uClibc Cross-Compilers 307
Retrieving the buildroot Package 308
Building a Cross-Compiler Using buildroot 309
Debugging and Resolving Toolchain Build Problems in buildroot 317
Building Cross-Compilers Manually 318

APPENDIX A Using GCC Compilers 321

Using Options with GCC Compilers 321
General Information Options 322
Controlling GCC Compiler Output 324
Controlling the Preprocessor 331
Modifying Directory Search Paths 333
Passing Options to the Assembler 335
Controlling the Linker 335
Enabling and Disabling Warning Messages 338
Adding Debugging Information 343
Customizing GCC Compilers 347
Customizing GCC Compilers Using Environment Variables 347
Customizing GCC Compilers with Spec Files and Spec Strings 349
Alphabetical GCC Option Reference 354

APPENDIX B Machine- and Processor-Specific Options for GCC 403

Alpha Options 403
Alpha/VMS Options 408
AMD x86-64 Options 408
AMD 29K Options 409
ARC Options 411
ARM Options 412
AVR Options 417
Blackfin Options 418
Clipper Options 419
Convex Options 419
CRIS Options 420
CRX Options 422
D30V Options 423
Darwin Options 423
FR-V Options 425
H8/300 Options 428
HP/PA (PA/RISC) Options 429
i386 and AMD x86-64 Options 431
IA-64 Options 437
Intel 960 Options 441
M32C Options 443
M32R Options 443
M680x0 Options 445
M68HC1x Options 447
M88K Options 448
MCore Options 450
MIPS Options 451
MMIX Options 458
MN10200 Options 459

MN10300 Options 459
MT Options 460
NS32K Options 460
PDP-11 Options 462
PowerPC (PPC) Options 463
RS/6000 Options 474
RT Options 474
S/390 and zSeries Options 475
SH Options 477
SPARC Options 479
System V Options 482
TMS320C3x/C4x Options 483
V850 Options 485
VAX Options 487
Xstormy16 Options 487
Xtensa Options 487

APPENDIX C Using GCC's Online Help 491

What Is GNU Info? 491
Getting Started, or Instructions for the Impatient 492
Getting Help 494
The Beginner's Guide to Using GNU Info 494
Anatomy of a GNU Info Screen 494
Moving Around in GNU Info 496
Performing Searches in GNU Info 498
Following Cross-References 499
Printing GNU Info Nodes 500
Invoking GNU Info 501
Stupid Info Tricks 502
Using Command Multipliers 502
Working with Multiple Windows 503

INDEX 505

About the Author

BILL VON HAGEN holds degrees in computer science, English writing, and art history. Bill has worked with Unix systems since 1982, during which time he has been a system administrator, writer, systems programmer, development manager, drummer, operations manager, content manager, and product manager. Bill has written a number of books including *The Ubuntu Bible*, *Hacking the TiVo*, *Linux Filesystems*, *Installing Red Hat Linux*, and *SGML for Dummies*; coauthored *Linux Server Hacks, Volume 2* and *Mac OS X Power User's Guide*; and contributed to several other books. Bill has written articles and software reviews for publications including *Linux Journal*, *Linux Magazine*, *Mac Tech*, *Linux Format* (UK), *Mac Format* (UK), and *Mac Directory*. He has also written extensive online content for CMP Media, Linux Planet, and Linux Today. An avid computer collector specializing in workstations, he owns more than 200 computer systems. You can contact Bill at `wvh@vonhagen.org`.

About the Technical Reviewer

GENE SALLY has been a Linux enthusiast for the past ten years, and for the past six he has channeled his enthusiasm through his employer, TimeSys, creating tools for embedded Linux engineers and helping them become more productive. Embedded development pushes the envelope of most technologies, Linux and GCC included, so Gene has had the opportunity to push these tools to their limits as he creates development tools and technologies for TimeSys' customers.

Acknowledgments

I'd like to thank Kurt Wall for his friendship and the opportunity to work with him on the first edition of this book, and Marta Justak, of Justak Literary Services, for her support and help with this book. I'd also like to thank Gene Sally for making this book far better than it could have been without him, and Richard Dal Porto, Keir Thomas, Jason Gilmore, Jennifer Whipple, Katie Stence, and others at Apress for their patience (!) and support for this second edition. In general, I'd like to thank GCC, emacs (the one true editor), Richard Stallman and the FSF, 50 million BSD fans (who can't be wrong), and Linux Torvalds and a cast of thousands for their contributions to computing as we know it today.

Without their foresight, philosophy, and hard work, this book wouldn't even exist. I'd especially like to thank rms for some way cool LMI hacks long ago.

Introduction

This book, *The Definitive Guide to GCC,* is about how to build, install, customize, use, and troubleshoot GCC version 4.x. GCC has long been available for most major hardware and operating system platforms and is often the preferred family of compilers.

As a general-purpose set of compilers, GCC produces high-quality, fast code. Due to its design, GCC is easy to port to different architectures, which contributes to its popularity. GCC, along with GNU Emacs, the Linux operating system, the Apache Web server, the Sendmail mail server, and the BIND DNS server, are showpieces of the free software world and proof that sometimes you **can** get a free lunch.

Why a Book About GCC?

I wrote this book, and you should read it, for a variety of reasons: it covers version 4.x; it is the only book that covers general GCC usage; and I would argue that it is better than GCC's own documentation. You will not find more complete coverage of GCC's features, quirks, and usage anywhere else in a single volume. There are no other up-to-date sources of information on GCC, excluding GCC's own documentation. GCC usually gets one or two chapters in programming books and only a few paragraphs in other more general titles.

GCC's existing documentation, although thorough and comprehensive, targets a programming-savvy reader. There's certainly nothing wrong with this approach, which is certainly the proper approach for advanced users, but GCC's own documentation leaves the great majority of its users out in the cold. Much of *The Definitive Guide to GCC* is tutorial and practical in nature, explaining why you use one option or why you should not use another one. In addition, explaining auxiliary tools and techniques that are relevant to GCC but not explicitly part of the package helps make this book a complete and usable guide and reference. Showing you how to use the compilers in the GCC family and related tools, and helping you get your work done are this book's primary goals.

Most people, including many long-time programmers, use GCC the way they learned or were taught to use it. That is, many GCC users treat the compiler as a black box, which means that they invoke it by using a small and familiar set of options and arguments they have memorized, shoving source files in one end, and then receiving a compiled, functioning program from the other end. With a powerful set of compilers such as GCC, there are indeed stranger (and more useful) things than were dreamed of in Computer Science 101. Therefore, another goal when writing *The Definitive Guide to GCC* was to reveal cool but potentially obscure options and techniques that you may find useful when building or using GCC and related tools and libraries.

Inveterate tweakers, incorrigible tinkerers, and the just plain adventurous among you will also enjoy the chance to play with the latest and greatest version of GCC and the challenge of bending a complex piece of software to your will, especially if you have instructions that show you how to do so with no negative impact on your existing system.

Why the New Edition?

I've written a new edition of this book for two main reasons: much has changed in GCC since the first edition of this book came out, and I wanted to talk about the other GCC compilers and related technologies such as cross-compilers and alternate C libraries. The GCC 4.x family of compilers is now available, providing a new optimization framework, many associated improvements to optimization in general, a new Fortran compiler, significant performance improvements for the C++ compiler, huge updates to the Java compiler, just-in-time compilation for Java, support for many new platforms, and enough new options in general to keep you updating Makefiles for quite a while. The first edition of this book focused on the C and C++ compilers in GCC, but enquiring minds want to know much more. This edition substantially expands the C++ coverage and adds information about using the Fortran, Java, and Objective-C compilers. No one has ever asked me about the Ada compiler, so I've still skipped that one. In addition, I've added information on using alternate C libraries and building cross-compilers that should make this book more valuable to its existing audience and (hopefully) attractive to an even larger one.

What You Will Learn

The Definitive Guide to GCC now provides a chapter dedicated to explaining how to use each of the C, C++, Fortran, and Java compilers. Information that is common to all of the compilers has been moved to Appendix A, so as not to repeat it everywhere and keep you from getting started with your favorite compiler. Similarly, information about building GCC has been moved to much later in the book, since most readers simply want to use the compilers that they find on their Linux and *BSD systems, not necessarily build them from scratch. However, if you want the latest and greatest version of GCC, you will learn how to download, compile, and install GCC from scratch, a poorly understood procedure that, until now, only the most capable and confident users have been willing to undertake.

The chapter on troubleshooting compilation problems has been expanded to make it easier than ever to discover problems in your code or the configuration or installation of your GCC compilers. If you're a traditional Makefile fan, the chapters on Libtool, Autoconf, and Automake will help you produce your Makefiles automatically, making it easier to package, archive, and distribute the source code for your projects. The chapters on code optimization, test coverage, and profiling have been expanded and updated to discuss the latest techniques and tools, helping you debug, improve, and test your code more extensively than ever. Finally, the book veers back to its focus for a more general audience by providing a complete summary of the GCC's command-line interface, a chapter on troubleshooting GCC usage and installation, and another chapter explaining how to use GCC's online documentation.

What You Need to Know

This is an end user's book intended for anyone using almost all of the GCC compilers (sorry, Ada fans). Whether you are a casual end user who only occasionally compiles programs, an intermediate user using GCC frequently but lacking much understanding of how it works, or a programmer seeking to exercise GCC to the full extent of its capabilities, you will find information in this book that you can use immediately. Because Linux and Intel x86 CPUs are so popular, I've assumed that most of you are using one version or another of the Linux operating system running on Intel x86 or compatible systems. This isn't critical—most of the material is GCC-specific, rather than being Linux- or Intel-specific, because GCC is largely independent of operating systems and CPU features in terms of its usage.

What do you need to know to benefit from this book? Well, knowing how to type is a good start because the GCC compilers are command-line compilers. (Though GCC compilers are integrated

into many graphical integrated development environments, that is somewhat outside the scope of this book.) You should therefore be comfortable with working in a command-line environment, such as a terminal window or a Unix or Linux console. You need to be computer literate, and the more experience you have with Unix or Unix-like systems, such as Linux, the better. If you have downloaded and compiled programs from source code before, you will be familiar with the terminology and processes discussed in the text. If, on the other hand, this is your first foray into working with source code, the chapters on building GCC and C libraries will get you up and running quickly. You do not need to be a programming wizard or know how to do your taxes in hexadecimal. Any experience that you have using a compiled programming language is gravy.

You should also know how to use a text editor, such as vi, pico, or Emacs, if you intend to type the listings and examples yourself in order to experiment with them. Because the source and binary versions of the GCC are usually available in some sort of compressed format, you will also need to know how to work with compressed file formats, usually gzipped tarballs, although the text will explain how to do so.

What *The Definitive Guide to GCC* Does Not Cover

As an end user's book on GCC, a number of topics are outside this book's scope. In particular, it is not a primer on C, C++, Fortran, or Java, although each chapter provides a consistent set of programming examples that I've used throughout the book. As discussed throughout this book, GCC is a collection of front-end, language-specific interfaces to a common back-end compilation engine. The list of compilers includes C, C++, Objective C, Fortran, Ada, and Java, among others. Compiler theory gets short shrift in this book, because I believe that most people are primarily interested in getting work done with GCC, not writing it. The Free Software Foundation has some excellent documents on GCC internals on its Web site, and it doesn't get much more definitive than that. That said, it is difficult to talk about using a compiler without skimming the surface of compiler theory and operation, so this book defines key terms and concepts as necessary while describing GCC's architecture and overall compilation workflow.

History and Overview of GCC

This section takes a more thorough look at what GCC is and does and includes the obligatory history of GCC. Because GCC is one of the GNU Project's premier projects, GCC's development model bears a closer look, so I will also show you GCC's development model, which should help you understand why GCC has some features and lacks other features, and how you can participate in its development.

What exactly is GCC? The tautological answer is that GCC is an acronym for the GNU Compiler Collection, formerly known as the GNU Compiler Suite, and also known as GNU CC and the GNU C Compiler. As remarked earlier, GCC is a collection of compiler front ends to a common back-end compilation engine. The list of compilers includes C, C++, Objective C, Fortran (now 95, formerly 77), and Java. GCC also has front ends for Pascal, Modula-3, and Ada 9X. The C compiler itself speaks several different dialects of C, including traditional and ANSI C. The C++ compiler is a true native C++ compiler. That is, it does not first convert C++ code into an intermediate C representation before compiling it, as did the early C++ compilers such as the Cfront "compiler" Bjarne Stroustrup first used to create C++. Rather, GCC's C++ compiler, g++, creates native executable code directly from the C++ source code.

GCC is an optimizing and cross-platform compiler. It supports general optimizations that can be applied regardless of the language in use or the target CPU and options specific to particular CPU families and even specific to a particular CPU model within a family of related processors. Moreover, the range of hardware platforms to which GCC has been ported is remarkably long. GCC supports platform and target submodels, so that it can generate executable code that will run on all members

of a particular CPU family or only on a specific model of that family. Table 1 provides a partial list of GCC's supported architectures, many of which you might never have heard of, much less used. Frankly, I haven't used (or even seen) all of them. For a more definitive list, see Appendix B, which summarizes architectures and processor-specific options for your convenience.

Considering the variety of CPUs and architectures to which GCC has been ported, it should be no surprise that you can configure it as a cross-compiler and use GCC to compile code on one platform that is intended to run on an entirely different platform. In fact, you can have multiple GCC configurations for various platforms installed on the same system and, moreover, run multiple GCC versions (older and newer) for the same CPU family on the same system.

Table 1. *Some of the Most Popular Processor Architectures Supported by GCC*

Architecture	Description
AMD29K	AMD Am29000 architectures
AMD64	64-bit AMD processors that are compatible with the Intel-32 architecture
ARM	Advanced RISC Machines architectures
ARC	Argonaut ARC processors
AVR	Atmel AVR microcontrollers
ColdFire	Motorola's latest generation of 68000 descendents
DEC Alpha	Compaq (neé Digital Equipment Corporation) Alpha processors
H8/300	Hitachi H8/300 CPUs
HP/PA	Hewlett-Packard PA-RISC architectures
Intel i386	Intel i386 (x86) family of CPUs
Intel i960	Intel i960 family of CPUs
M32R/D	Mitsubishi M32R/D architectures
M68K	The Motorola 68000 series of CPUs
M88K	Motorola 88K architectures
MCore	Motorola M*Core processors
MIPS	MIPS architectures
MN10200	Matsushita MN10200 architectures
MN10300	Matsushita MN10300 architectures
NS32K	National Semiconductor NS3200 CPUs
RS/6000 and PowerPC	IBM RS/6000 and PowerPC architectures
S390	IBM processors used in zSeries and System z mainframe
SPARC	Sun Microsystems family of SPARC CPUs
SH3/4/5	Super Hitachi 3, 4, and 5 family of processors
TMS320C3x/C4x	Texas Instruments TMS320C3x and TMS320C4x DSPs

GCC's History

GCC, or rather, the idea for it, actually predates the GNU Project. In late 1983, just before he started the GNU Project, Richard M. Stallman, president of the Free Software Foundation and originator of the GNU Project, heard about a compiler named the Free University Compiler Kit (known as VUCK) that was designed to compile multiple languages, including C, and to support multiple target CPUs. Stallman realized that he needed to be able to bootstrap the GNU system and that a compiler was the first strap he needed to boot. So he wrote to VUCK's author asking if GNU could use it. Evidently, VUCK's developer was uncooperative, responding that the university was free but that the compiler was not. As a result, Stallman concluded that his first program for the GNU Project would be a multilanguage, cross-platform compiler. Undeterred and in true hacker fashion, desiring to avoid writing the entire compiler himself, Stallman eventually obtained the source code for Pastel, a multiplatform compiler developed at Lawrence Livermore National Laboratory. He added a C front end to Pastel and began porting it to the Motorola 68000 platform, only to encounter a significant technical obstacle: the compiler's design required many more megabytes of stack space than the 68000-based Unix system supported. This situation forced him to conclude that he would have to write a new compiler, starting from ground zero. That new compiler eventually became GCC.

Although it contains none of the Pastel source code that originally inspired it, Stallman did adapt and use the C front end he wrote for Pastel. As a starting point for GCC's optimizer, Stallman also used PO, a portable peephole optimizer that performed optimizations generally done by high-level optimizers, in addition to low-level peephole optimizers. GCC (and PO's successor, vpo) still uses RTL (register transfer language) as an intermediate format for the optimizer. Development of this primordial GCC proceeded slowly through the 1980s, because, as Stallman writes in his description of the GNU Project (`http://www.gnu.org/gnu/the-gnu-project.html`), "first, [he] worked on GNU Emacs."

During the 1990s, GCC development split into two, perhaps three, branches. While the primary GCC branch continued to be maintained by the GNU Project, a number of other developers, primarily associated with Cygnus Solutions, began releasing a version of GCC known as EGCS (Experimental [or Enhanced] GNU Compiler Suite). EGCS was intended to be a more actively developed and more efficient compiler than GCC, but was otherwise effectively the same compiler because it closely tracked the GCC code base and EGCS enhancements were fed back into the GCC code base maintained by the GNU Project. Nonetheless, the two code bases were separately maintained. In April 1999, GCC's maintainers, the GNU Project, and the EGCS steering committee formally merged. At the same time, GCC's name was changed to the GNU Compiler Collection and the separately maintained (but, as noted, closely synchronized) code trees were formally combined, ending a long fork and incorporating the many bug fixes and enhancements made in EGCS into GCC. This is why EGCS is often mentioned, though it is officially defunct.

Other historical variants of GCC include the Pentium Compiler Group (PCG) project's own version of GCC, PGCC. PGCC was a Pentium-specific version that was intended to provide the best possible support for features found in Intel's Pentium-class CPUs. During the period of time that EGCS was separately maintained, PGCC closely tracked the EGCS releases. The reunification of EGCS and GCC seems to have halted PGCC development because, at the time of this writing, the PCG project's last release was 2.95.2.1, dated December 27, 2000. For additional information, visit the PGCC project's Web site at `http://www.goof.com/pcg/`.

At the time that this book was written, GCC 4.2 was about to become available. The latest officially released version of the GCC 3.x line of compilers is 3.4.5. Other significant milestone compilers are the 2.95.x compilers, which were widely hacked to produce code for a variety of embedded systems and which are still widely available.

Who Maintains GCC?

Formally, GCC is a GNU Project, which is directed by the FSF. The FSF holds the copyright on the compilers, and licenses the compilers under the terms of the GPL. Either individuals or the FSF hold

the copyrights on other components, such as the runtime libraries and test suites, and these other components are licensed under a variety of licenses for free software. For information on the licensing of any FSF package see the file LICENSE that is provided with its source code distribution. The FSF also handles the legal concerns of the GCC project. So much for the administrivia.

On the practical side, a cast of dozens maintains GCC. GCC's maintainers consist of a formally organized steering committee and a larger, more loosely organized group of hackers scattered all over the Internet. The GCC steering committee, as of August 2001, is made up of 14 people representing various communities in GCC's user base who have a significant stake in GCC's continuing and long-term survival, including kernel hackers, Fortran users, and embedded systems developers. The steering committee's purpose is, to quote its mission statement, "to make major decisions in the best interests of the GCC project and to ensure that the project adheres to its fundamental principles found in the project's mission statement." These "fundamental principles" include the following:

- Supporting the goals of the GNU Project
- Adding new languages, optimizations, and targets to GCC
- More frequent releases
- Greater responsiveness to consumers, the large user base that relies on the GCC compiler
- An open development model that accepts input and contributions based on technical merit

The group of developers that work on GCC includes members of the steering committee and, according to the contributors list on the GCC project home page, more than 100 other individuals across the world. Still, others not specifically identified as contributors have contributed to GCC development by sending in patches, answering questions on the various GCC mailing lists, submitting bug reports, writing documentation, and testing new releases.

Who Uses GCC?

GCC's user base is large and varied. Given the nature of GCC and the loosely knit structure of the free software community, though, no direct estimate of the total number of GCC users is possible. A direct estimate, based on standard metrics, such as sales figures, unit shipments, or license purchases, is virtually impossible to derive because such numbers simply do not exist. Even indirect estimates, based, for example, on the number of downloads from the GNU Web and FTP sites, would be questionable because the GNU software repository is mirrored all over the world.

More to the point, I submit that quantifying the number of GCC users is considerably less important and says less about GCC users than examining the scope of GCC's usage and the number of processor architectures to which it has been ported. For example, GCC is the standard compiler shipped in every major and most minor Linux distributions. GCC is also the compiler of choice for the various BSD operating systems (FreeBSD, NetBSD, OpenBSD, and so on). Thanks initially to the work of DJ Delorie, GCC works on most modern DOS versions, including MS-DOS from Microsoft, PC-DOS from IBM, and DR-DOS. Indeed, Delorie's work resulted in ports of most of the GNU tools for DOS-like environments. Cygnus Solutions, now owned by Red Hat, Inc., created a GCC port for Microsoft Windows users. Both the DOS and Windows ports offer complete and free development environments for DOS and Windows users.

The academic computing community represents another large part of GCC's user base. Vendors of hardware and proprietary operating systems typically provide compiler suites for their products as a so-called value-added service, that is, for an additional, often substantial, charge. As free software, GCC represents a compelling, attractive alternative to computer science departments faced with tight budgets. GCC also appeals to the academic world because it is available in source code form, giving students a chance to study compiler theory, design, and implementation. GCC is also widely used by nonacademic customers of hardware and operating system vendors who want to

reduce support costs by using a free, high-quality compiler. Indeed, if you consider the broad range of hardware to which GCC has been ported, it becomes quite clear that GCC's user base is composed of the broadest imaginable range of computer users.

In general, my favorite response from any reader of this book to the question of who uses GCC is "I do."

Are There Alternatives?

What alternatives to GCC exist? As framed, this question is somewhat difficult to answer. Remember that GCC is the GNU Compiler Collection, a group of language-specific compiler front ends using a common back-end compilation engine, and that GCC is free software. So if you rephrase the question to "what free compiler suites exist as alternatives to GCC?" the answer is "very few."

As mentioned earlier, the Pentium Compiler Group created PGCC, a version of GCC, that was intended to extend GCC's ability to optimize code for Intel's Pentium-class CPUs. Although PGCC development seems to have stalled since the EGCS/GCC schism ended, the PGCC Web site still exists (although it, too, has not been modified recently).

If you remove the requirement that the alternative be free, you have many more options. Many hardware vendors and most operating system vendors will be happy to sell you compiler suites for their respective hardware platforms or operating systems, but the cost can be prohibitive. Some third-party vendors exist that provide stand-alone compiler suites. One such vendor is The Portland Group (`http://www.pgroup.com/`), which markets a set of high-performance, parallelizing compiler suites supporting Fortran, C, and C++. Absoft Corporation also offers a well-regarded compiler suite supporting Fortran 77, Fortran 95, C, and C++. Visit its Web site at `http://www.absoft.com/` for additional information. Similarly, Borland has a free C/C++ compiler available. Information on Borland's tools can be found on its Web site at `http://www.borland.com/`. Intel and Microsoft also sell very good compilers. And they are not that expensive.

Conversely, if you dispense with the requirement that alternatives be collections or suites, you can select from a rich array of options. A simple search for the word *compilers* at Yahoo! generates more than 120 Web sites showcasing a variety of single-language compilers, including Ada, Basic, C and C++, COBOL, Forth, Java, Logo, Modula-2 and Modula-3, Pascal, Prolog, and Smalltalk. If you are looking for alternatives to GCC, a good place to start your search is the compilers.net Web page at `http://www.compilers.net/`.

So much for a look at alternatives to GCC. This is a book about GCC, after all, so I hope that you'll forgive me for largely leaving you on your own when it comes to finding information about other compilers. Some chapters of this book, such as the chapter on the new GCC Fortran compiler, gfortran, discuss alternatives because of the huge number of Fortran variants out there, but by and large, GCC is the right solution to your compilation problems.

CHAPTER 1

Using GCC's C Compiler

This chapter's goal is to get you comfortable with typical usage of the GNU Compiler Collection's C compiler, gcc. This chapter focuses on those command-line options and constructs that are specific to GCC's C compiler. Options that can generally be used with any GCC compiler are discussed in Appendix A. Throughout this chapter, as throughout this book, I'll differentiate between GCC (the GNU Compiler Collection) and gcc, the C compiler that is provided as part of GCC.

This chapter explains how to tell gcc which dialect of C it should expect to encounter in your source files, from strict ANSI/ISO C to classic Kernighan and Ritchie (K&R) C. It also explains the variety of special-purpose constructs that are supported by gcc and how to invoke and use them. It concludes by discussing using gcc to compile Objective C applications and discusses specific details of the GNU Objective C runtime environment.

GCC Option Refresher

Appendix A discusses the options that are common to all of the GCC compilers and how to customize various portions of the compilation process. However, I'm not a big fan of making people jump around in a book for information. For that reason, this section provides a quick refresher of basic GCC compiler usage as it applies to the gcc C compiler. For detailed information, see Appendix A. If you are new to gcc and just want to get started quickly, you're in the right place.

The gcc compiler accepts both single-letter options, such as `-o`, and multiletter options, such as `-ansi`. Because it accepts both types of options you cannot group multiple single-letter options together as you may be used to doing in many GNU and Unix/Linux programs. For example, the multiletter option `-pg` is not the same as the two single-letter options `-p -g`. The `-pg` option creates extra code in the final binary that outputs profile information for the GNU code profiler, gprof. On the other hand, the `-p -g` options generate extra code in the resulting binary that produces profiling information for use by the prof code profiler (`-p`) and causes gcc to generate debugging information using the operating system's normal format (`-g`).

Despite its sensitivity to the grouping of multiple single-letter options, you are generally free to mix the order of options and compiler arguments on the gcc command line. That is, invoking gcc as

```
gcc -pg -fno-strength-reduce -g myprog.c -o myprog
```

has the same result as

```
gcc myprog.c -o myprog -g -fno-strength-reduce -pg
```

I say that you are generally free to mix the order of options and compiler arguments because, in most cases, the order of options and their arguments does not matter. In some situations, order does matter if you use several options of the same kind. For example, the `-I` option specifies the directory or directories to search for include files. So if you specify `-I` several times, gcc searches the listed directories in the order specified.

Compiling a single source file, myprog.c, using gcc is easy—just invoke gcc, passing the name of the source file as the argument.

```
$ gcc myprog.c
$ ls -l
```

```
-rwxr-xr-x      1 wvh    users           13644 Oct 5 16:17 a.out
-rw-r--r--      1 wvh    users           220   Oct 5 16:17 myprog.c
```

By default, the result on Linux and Unix systems is an executable file named a.out in the current directory, which you execute by typing **./a.out**. On Cygwin systems, you will wind up with a file named a.exe that you can execute by typing either **./a** or **./a.exe**.

To define the name of the output file that gcc produces, use the `-o` option, as illustrated in the following example:

```
$ gcc myprog.c -o runme
$ ls -l
```

```
-rw-r--r--    1 wvh    users       220   Oct  5 16:17 myprog.c
-rwxr-xr-x    1 wvh    users       13644 Oct  5 16:28 runme
```

If you are compiling multiple source files using gcc, you can simply specify them all on the gcc command line, as in the following example, which leaves the compiled and linked executable in the file named showdate:

```
$ gcc showdate.c helper.c –o showdate
```

If you want to compile these files incrementally and eventually link them into a binary, you can use the `–c` option to halt compilation after producing an object file, as in the following example:

```
$ gcc -c showdate.c
$ gcc -c helper.c
$ gcc showdate.o helper.o –o showdate
$ ls -l
```

```
total 124
-rw-r--r--    1 wvh    users        210   Oct  5 12:42 helper.c
-rw-r--r--    1 wvh    users        45    Oct  5 12:29 helper.h
-rw-r--r--    1 wvh    users        1104  Oct  5 13:50 helper.o
-rwxr-xr-x    1 wvh    users        13891 Oct  5 13:51 showdate
-rw-r--r--    1 wvh    users        208   Oct  5 12:44 showdate.c
-rw-r--r--    1 wvh    users        1008  Oct  5 13:50 showdate.o
```

■ **Note** All of the GCC compilers "do the right thing" based on the extensions of the files provided on any GCC command line. Mapping file extensions to actions (for example, understanding that files with `.o` extensions only need to be linked) is done via the GCC specs file. Prior to GCC version 4, the specs file was a stand-alone text file that could be modified using a text editor; with GCC 4 and later, the specs file is built-in and must be extracted before it can be modified. For more information about working with the specs file, see the section "Customizing GCC Using Spec Strings" in Appendix A.

It should be easy to see that a project consisting of more than a few source code files would quickly become exceedingly tedious to compile from the command line, especially after you start adding search directories, optimizations, and other gcc options. The solution to this command-line tedium is the make utility, which is not discussed in this book due to space constraints (although it is touched upon in Chapter 8).

Compiling C Dialects

The gcc compiler supports a variety of dialects of C via a range of command-line options that enable both single features and ranges of features that are specific to particular variations of C. Why bother, you ask? The most common reason to compile code for a specific dialect of C is for portability. If you write code that might be compiled with several different tools, you can check for that code's adherence to a given standard using GCC support for various dialects and standards. Verifying adherence to various standards is one method developers use to reduce the risk of running into compile-time and runtime problems when code is moved from one platform to another, especially when the new platform was not considered when the program was originally written.

What then is wrong with vanilla ISO/ANSI C? Nothing that has not been corrected by officially ordained corrections. The original ANSI C standard, prosaically referred to as C89, is officially known as ANSI X3.159-1989. It was ratified by ANSI in 1989 and became an ISO standard, ISO/IEC9989:1990 to be precise, in 1990. Errors and slight modifications were made to C89 in technical corrigenda published in 1994 and 1996. A new standard, published in 1999, is known colloquially as C99 and officially as ISO/IEC9989:1999. The freshly minted C99 standard was amended by a corrigendum issued in 2001. This foray into the alphabet soup of standards explains why options are available for supporting multiple dialects of C. I'll explain how to use them a little later in this section.

In addition to the subtle variations that exist in standard C, some of the gcc C dialect options enable you to select the degree to which gcc complies with the standard. Other options enable you to select which C features you want. There is even a switch that enables limited support for traditional (pre-ISO, pre-ANSI) C. But enough discussion. Table 1-1 lists and briefly describes the options that control the C dialect to which gcc adheres during compilation.

Table 1-1. *C Dialect Command-Line Options*

Option	Description
`-ansi`	Supports all ISO C89 features and disables GNU extensions that conflict with the C89 standard.
`-aux-info` *file*	Saves prototypes and other identifying information about functions declared in a translation unit to the file identified by *file*.
`-fallow-single-precision`	Prevents promotion of single-precision operations to double-precision.

Table 1-1. *C Dialect Command-Line Options (Continued)*

Option	Description
`-fbuiltin`	Recognizes built-in functions that lack the `__builtin_` prefix.
`-fcond-mismatch`	Allows mismatched types in the second and third arguments of conditional statements.
`-ffreestanding`	Claims that compilation takes place in a freestanding (unhosted) environment. Freestanding means that the environment includes all of the library functions required to operate without loading or referencing external code. Currently, freestanding implementations provide all of the functions identified in `<float.h>`, `<limits.h>`, `<stdarg.h>`, and `<stddef.h>`. Freestanding 64-bit code also requires the functions identified in `<iso646.h>`. Freestanding C99-compliant code also requires anything referenced in `<stdbool.h>` and `<stdint.h>`. The Linux kernel is a good example of a freestanding environment.
`-fhosted`	Claims that compilation takes place in a hosted environment, which means that external functions can be loaded from libraries such as the standard C library. This is the default value for GCC compilation. Programs that use external libraries (such as most applications) are good examples of applications that compile and execute in a hosted environment.
`-fno-asm`	Disables use of `asm`, `inline`, and `typeof` as keywords, allowing their use as identifiers.
`-fno-builtin`	Ignores built-in functions that lack the `__builtin_` prefix.
`-fno-signed-bitfields`	Indicates that bit fields of undeclared type are to be considered unsigned.
`-fno-signed-char`	Keeps the `char` type from being signed, as in the type `signed char`.
`-fno-unsigned-bitfields`	Indicates that bit fields of undeclared type are to be considered signed.
`-fno-unsigned-char`	Keeps the `char` type from being unsigned, as in the type `unsigned char`.
`-fshort-wchar`	Forces the type `wchar_t` to be `short unsigned int`.
`-fsigned-bitfields`	Indicates that bit fields of undeclared type are to be considered signed.
`-fsigned-char`	Permits the `char` type to be signed, as in the type `signed char`.
`-funsigned-bitfields`	Indicates that bit fields of undeclared type are to be considered unsigned.
`-funsigned-char`	Permits the `char` type to be unsigned, as in the type `unsigned char`.
`-fwritable-strings`	Permits strong constants to be written and stores them in the writable data segment.
`-no-integrated-cpp`	Invokes an external C preprocessor instead of the integrated preprocessor.
`-std=`*value*	Sets the language standard to value (`c89`, `iso9899:1990`, `iso9989:199409`, `c99`, `c9x`, `iso9899:1999`, `iso9989:199x`, `gnu89`, `gnu99`).

Table 1-1. *C Dialect Command-Line Options (Continued)*

Option	Description
`-traditional`	Supports a limited number of traditional (K&R) C constructs and features.
`-traditional-cpp`	Supports a limited number of traditional (K&R) C preprocessor constructs and features.
`-trigraphs`	Enables support for C89 trigraphs.

Sufficiently confused? Believe it or not, it breaks down more simply than it seems. To begin with, throw out `-aux-info` and `-trigraphs`, because you are unlikely to ever need them. Similarly, you are advised to not use `-no-integrated-cpp` because its semantics are subject to change and may, in fact, be removed from future versions of GCC. If you want to use an external preprocessor, use the `CPP` environment variable discussed in Appendix A or the corresponding `make` variable. Likewise, unless you are working with old code that assumes it can be scribbled into constant strings, do not use `-fwritable-strings`. After all, constant strings should be constant—if you are scribbling on them, they are variables, so just create a variable. To be fair, however, early C implementations allowed writable strings (primarily to limit stack space consumption), so this option exists to enable you to compile legacy code.

The various flags for signed or unsigned types exist to help you work with code that makes assumptions of the signedness of chars and bit fields. In the case of the char flags (`-fsigned-char`, `-funsigned-char`, and their corresponding negative forms), each machine has a default `char` type, which is either signed or unsigned. That is, given the statement

```
char c;
```

you might wind up with a `char` type that behaves like a `signed char` or an `unsigned char` on a given machine. If you pass gcc the `-fsigned-char` option, it will assume that all such unspecified declarations are equivalent to the statement

```
signed char c;
```

The converse applies if you pass gcc the `-funsigned-char` option. The purpose of these flags (and their negative forms) is to allow code that assumes the default machine `char` type is, say, like an unsigned char (that is, it performs operations on `char` types that assume an unsigned char), to work properly on a machine whose default `char` type is like a signed char. In this case, you would pass gcc the `-funsigned-char` option. A similar situation applies to the bit field–related options. In the case of bit fields, however, if the code does not specifically use the signed or unsigned keyword, gcc assumes the bit field is signed.

■ **Note** Truly portable code should not make such assumptions—that is, if you know you need a specific type of variable, say an unsigned char, you should declare it as such rather than using the generic type and making assumptions about its signedness that might be valid on one architecture but not on another.

You will rarely ever need to worry about the `-fhosted` and `-ffreestanding` options, but for completeness' sake, I'll explain what they mean and why they are important. In the world of C standards, an environment is either hosted or freestanding. A hosted environment refers to one in which the complete standard library is present and in which the program startup and termination occur via

a `main()` function that returns `int`. In a freestanding environment, on the other hand, the standard library may not exist and program startup and termination are implementation-defined. The implication of the difference is just this: in a freestanding implementation (when invoked with `-ffreestanding`), the gcc compiler makes very few assumptions about the meaning of function names that exist in the standard library. So, for example, the `ctime()` function is meaningless to gcc in freestanding mode. In hosted mode, which is the default, on the other hand, you can rely on the fact that the `ctime()` function behaves as defined in the C89 (or C99) standard.

■ **Note** This discussion simplifies the distinction between freestanding and hosted environments and ignores the distinction the ISO standard draws between conforming language implementations and program environments.

Now, about those options that control to which standards GCC adheres. Taking into account the command-line options I've already discussed, you are left with `-ansi`, `-std`, `-traditional`, `-traditional-cpp`, `-fno-asm`, `-fbuiltin`, and `-fno-builtin`. Here again, we can simplify matters somewhat. The `–traditional` option enables you to use features of pre-ISO C, and implies `-traditional-cpp`. These traditional C features include writable string constants (as with `-fwritable-strings`), the use of certain C89 keywords as identifiers (`inline`, `typeof`, `const`, `volatile`, and `signed`), and global extern declarations. You can see by looking at Table 1-1 that `-traditional` also implies `-fno-asm`, because `-fno-asm` disables the use of the `inline` and `typeof` keywords, such as `-traditional`, and also the `asm` keyword. In K&R C, these keywords could be used as identifiers.

The `-fno-builtin` flag disables recognition of built-in functions that do not begin with the `__builtin_` prefix. What exactly is a built-in function? Built-in functions are versions of functions in the standard C library that are implemented internally by gcc. Some built-ins are used internally by gcc, and only by gcc. These functions are subject to change, so they should not be used by non-GCC developers. Most of gcc's built-ins, though, are optimized versions of functions in the standard libraries, intended as faster and more efficient replacements of their externally defined cousins. You normally get these benefits for free because gcc uses its own versions of, say, `alloca()` or `memcpy()` instead of those defined in the standard C libraries. Invoking the `-fno-builtin` option disables this behavior. The GCC info pages document the complete list of gcc's built-in functions.

The `-ansi` and `-std` options, which force varying degrees of stricter adherence to published C standards documents, imply `-fno-builtin`. As Table 1-1 indicates, `-ansi` causes gcc to support all features of ISO C89 and turns off GNU extensions that conflict with this standard. To be clear, if you specify `-ansi`, you are selecting adherence to the C89 standard, not the C99 standard. The options `-std=c89` or `-std=iso9899:1990` have the same effect as `-ansi`. However, using any of these three options does not mean that gcc starts behaving as a strict ANSI compiler because GCC will not emit all of the diagnostic messages required by the standard. To obtain all of the diagnostic messages, you must also specify the options `-pedantic` or `-pedantic-errors`. If you want the diagnostics to be considered warnings, use `-pedantic`. If you want the diagnostics to be considered errors and thus to terminate compilation, use `-pedantic-errors`.

To select the C99 standard, use the option `-std=c99` or `-std-iso9989:1999`. Again, to see all of the diagnostic messages required by the C99 standard, use `-pedantic` or `-pedantic-errors` as previously described. To completely confuse things, the GNU folks provide arguments to the `-std` option that specify an intermediate level of standards compliance. Lacking explicit definition of a C dialect, gcc defaults to C89 mode with some additional GNU extensions to the C language. You can explicitly request this dialect by specifying `-std=gnu89`. If you want C99 mode with GNU extensions, you should specify, you guessed it, `-std=gnu99`. The default compiler dialect will change to `-std=gnu99` after gcc's C99 support has been completed.

What does turning on standards compliance do? Depending on whether you select C89 or C99 mode, the effects of `-ansi` or `-std=`*value* include

- Disabling the `asm` and `typeof` keywords
- Enabling trigraphs and digraphs
- Disabling predefined macros, such as `unix` or `linux`, that identify the type of system in use
- Disabling the use of // single-line comments in C code in C89 mode (C99 permits // single-line comments)
- Defining the macro `__STRICT_ANSI__` used by header files and functions to enable or disable certain features that are or are not C89-compliant
- Disabling built-in functions that conflict with those defined by the ISO standard
- Disabling all GNU extensions that conflict with the standard (GNU extensions to C are discussed in detail later in this chapter.)

Exploring C Warning Messages

A warning is a diagnostic message that identifies a code construct that might potentially be an error. GCC's C compiler also emits diagnostic messages when it encounters code or usage that looks questionable or ambiguous. Appendix A provides a discussion of the most commonly used options related to warning messages. This section explores some of the most common warning messages and related options as they apply to C programs compiled using the GCC C compiler, gcc. See Table A-7 in Appendix A for a complete list of the warning-related options available in GCC compilers. This section highlights the use of various warning options as they relate to compiling programs written in the C language.

When you compile C applications using the `–pedantic` option, you can also specify `-std=`*`version`* to indicate against which version of the standard the code should be evaluated. It is a mistake, however to use `-pedantic` (and, by extension, `-pedantic-errors`), even in combination with `-ansi`, to see if your programs are strictly conforming ISO C programs. This is the case because these options only emit diagnostic messages in situations for which the standard requires a diagnostic—the effort to implement a strict ISO parser would be enormous. In general, the purpose of `-pedantic` is to detect gratuitously noncompliant code, disable GNU extensions, and reject C++ and traditional C features not present in the standard.

Another reason that you cannot use `-pedantic` to check for strict ISO compliance is that `-pedantic` disregards the alternate keywords whose names begin and end with __ and expressions following `__extension__`. As a rule, these two exceptions do not apply to user programs because the alternate keywords and the `__extension__` usage is limited (or should be) to system header files, which application programs should never directly include.

■ **Note** Alternate keywords are discussed later in this chapter in the section titled "Declaring Function Attributes."

A typical source of trouble in C code emerges from the use of functions in the `printf()`, `scanf()`, `strftime()`, and `strfmon()` families. These calls use format strings to manipulate their arguments. Common problems with these function groups include type mismatches between the format strings and their associated arguments, too many or too few arguments for the supplied format strings, and potential security issues with format strings (known generically as format string exploits). The `-Wformat` warnings help you to identify and solve these problems.

Specifying the `-Wformat` option causes gcc to check all calls to `printf()`, `scanf()`, and other functions that rely on format strings, making sure that the arguments supplied have types appropriate to the format string and that the requested conversions, as specified in the format string, are sensible.

Moreover, if you use `-pedantic` with `-Wformat`, you can identify format features not consistent with the selected standard.

Consider the program shown in Listing 1-1 that uses `printf()` to print some text to the screen.

Listing 1-1. *Source Code for the Sample printme.c Application*

```
#include <stdio.h>
#include <limits.h>

int main (void)
{
        unsigned long ul = LONG_MAX;
        short int si = SHRT_MIN;

        printf ("%d\n", ul);
        printf ("%s\n", si);

        return 0;
}
```

There are a couple of problems here. The first `printf()` statement prints the value of the unsigned long variable `ul` using an `int` format string (`%d`). The second `printf()` statement prints the short `int` variable `si` using the `%s` format string (that is, as a string). Despite these problems, gcc compiles the program without complaint. It even sort of runs.

```
$ gcc printme.c -o printme
$ ./printme
```

```
2147483647
Segmentation fault
```

■ **Note** Depending on your system, this program may not generate a segmentation fault.

Well, that was ugly. On my test system, the program even crashed because the `%s` formatting option tried to use the `short int si` as a pointer to the beginning of a character string, which it is not. Now, add the `-Wformat` option and see what happens:

```
$ gcc printme.c -o printme -Wformat
```

```
printme.c: In function 'main':
printme.c:9: warning: int format, long int arg (arg 2)
printme.c:10: warning: format argument is not a pointer (arg 2)
$ ./printme
2147483647
Segmentation fault
```

■ **Note** Depending on the compiler version, your warning output might be slightly different.

This time, the compiler complains about the mismatched types and helpfully tells us where I can find the problems. The program still compiles, but I can use the `-Werror` option to convert the warning to a hard error that terminates compilation:

```
$ gcc printme.c -o printme -Wformat -Werror
```

```
cc1: warnings being treated as errors
printme.c: In function 'main':
printme.c:9: warning: int format, long int arg (arg 2)
printme.c:10: warning: format argument is not a pointer (arg 2)
```

This time, compilation stops after the errors are detected. Once I fix the mismatches, the program compiles and runs properly. Listing 1-2 shows the corrected program.

Listing 1-2. *Printme.c After Corrections*

```
#include <stdio.h>
#include <limits.h>

int main (void)
{
        unsigned long ul = LONG_MAX;
        short int si = SHRT_MIN;

        printf ("%ld\n", ul);
        printf ("%d\n", si);

        return 0;
}
$ gcc printme.c -o printme -Wformat -Werror
$ ./printme
2147483647
-32768
$
```

Much better, yes?

For more control over GCC's format checking, you can use the options `-Wno-format-y2k`, `-Wno-format-extra-args`, and `-Wformat-security` (only `-Wformat` is included in the `-Wall roll-up` option). By default, if you supply more arguments than are supported by the available format strings, the C standard says that the extra arguments are ignored. GCC accordingly ignores the extra arguments unless you specify `-Wformat`. If you want to use `-Wformat` and ignore extra arguments, specify `-Wno-format-extra-args`.

The `-Wformat-security` option is quite interesting. Format string exploits have become popular in the world of blackhats in the past few years. Specifying `-Wformat` and `-Wformat-security` displays warnings about format functions that represent potential security breaches. The current implementation covers `printf()` and `scanf()` calls in which the format string is not a string literal and in which there are no format arguments, such as `printf (var);`. Such code is problematic if the format string comes from untrusted input and contains `%n`, which causes `printf()` to write to system memory. Recall that the conversion specifier `%n` causes the number of characters written to be stored in the `int` pointer (`int *` or a variant thereof)—it is this memory write that creates the security issue.

The options that fall under the `-Wunused` category (see Table A-7 in Appendix A for a complete list) are particularly helpful. In optimization passes, gcc does a good job of optimizing away unused objects, but if you disable optimization, the unused cruft bloats the code. More generally, unused

objects and unreachable code (detected with `-Wunreachable-code`) are often signs of sloppy coding or faulty design. My own preference is to use the plain `-Wunused` option, which catches all unused objects. If you prefer otherwise, you can use any combination of the five specific `-Wunused-*` options.

Listing 1-3 is a short example of a program with an unused variable.

Listing 1-3. *Source Code for the Sample unused.c Application*

```
int main (void)
{
        int i = 10;
        return 0;
}
```

As you can see, the program defines the `int` variable `i`, but never does anything with it. Here is GCC's output when compiling `unused.c` with no options:

```
$ gcc unused.c
```

Well, perhaps I really should have written, "Here is the lack of GCC's output when compiling `unused.c` with no options," because adding `-ansi -pedantic` does not change the compiler's output. However, here is GCC's complaint when I use `-Wunused`:

```
$ gcc -Wunused unused.c
unused.c: In function 'main':
unused.c:3: warning: unused variable 'i'
```

Each of the warning options discussed in this section results in similar output that identifies the translation unit and line number in which a problem is detected, and a brief message describing the problem. (In the context of program compilation, a translation unit is a separately processed portion of your code, which usually means the code contained in a single file. It means something completely different when translating natural languages.) I encourage you to experiment with the warning options. Given the rich set of choices, you can debug and improve the overall quality and readability of your code just by compiling with a judiciously chosen set of warning options.

GCC's C and Extensions

As I remarked at the beginning of this chapter, GCC's C compiler or, rather, GNU C, provides language features that are not available in ANSI/ISO standard C. The following is an up-to-date summary of the most interesting of these features:

- *Local labels*: Write expressions containing labels with local scope
- *Labels as values*: Obtain pointers to labels and treat labels as computed `goto`s
- *Nested functions*: Define functions within functions
- *Constructing calls*: Dispatch a call to another function
- *Typeof*: Determine an expression's type at runtime using `typeof`
- *Zero- and variable-length arrays*: Declare zero-length arrays
- *Variable-length arrays*: Declare arrays whose length is computed at runtime
- *Variadic macros*: Define macros with a variable number of arguments
- *Subscripting*: Subscripts any array, even if not an lvalue
- *Pointer arithmetic*: Performs arithmetic on void pointers and on function pointers
- *Initializers*: Assign initializers using variable values

- *Designated initializers*: Label elements of initializers
- *Case ranges*: Represent syntactic sugar in switch statements
- *Mixed declarations*: Mix declarations and code
- *Function attributes*: Declare that a function has no side effects or never returns
- *Variable attributes*: Specify attributes of variables
- *Type attributes*: Specify attributes of types
- *Inline*: Defines inline functions (as fast as macros)
- *Function names*: Display the name of the current function
- *Return addresses*: Obtain a function's return address or frame address
- *Pragmas*: Use GCC-specific pragmas
- *Unnamed fields*: Embed unnamed struct and union fields within named `structs` and `unions`

Locally Declared Labels

Each statement expression is a scope in which local labels can be declared. A *local label* is simply an identifier; you can jump to it with an ordinary `goto` statement, but only from within the statement expression it belongs to. A local label declaration looks like this:

```
__label__ alabel;
```

or

```
__label__ alabel, blabel, ...;
```

The first statement declares a local label named `alabel`, and the second one declares two labels named `alabel` and `blabel`. Local label declarations must appear at the beginning of the statement expression, immediately following the opening brace ({) and before any ordinary declarations.

A label declaration defines a label name, but does not define the label itself. You do this using `name` within the statement's expression to define the contents of the label named `name`. For example, consider the following code:

```
int main(void)
{
       __label__ something;
       int foo;

       foo=0;
       goto something;
       {
               __label__ something;
               goto something;
               something:
                       foo++;
       }
       something:
               return foo;
}
```

This code declares two labels named `something`. The label in the outer block must be distinguished from the label in the local block. Local label declaration means that the label must still be unique with respect to the largest enclosing block. So if you put a label in a macro, you need to give it a

mangled name to avoid clashing with a name the user has defined. The local label feature is useful because statement expressions are often used in macros. If the macro contains nested loops, a goto can be useful for breaking out of them. However, an ordinary label whose scope is the whole function cannot be used; if the macro can be expanded several times in one function, the label will be multiply-defined in that function.

Local labels avoid this problem:

```
#define SEARCH(array, target)                          \
({                                                     \
    __label__ found;                                   \
    typeof (target) _SEARCH_target = (target);         \
    typeof (*(array)) *_SEARCH_array = (array);        \
    int i, j;                                          \
    int value;                                         \
    for (i = 0; i < max; i++)                          \
        for (j = 0; j < max; j++)                      \
            if (_SEARCH_array[i][j] == _SEARCH_target) \
                { value = i; goto found; }             \
        value = -1;                                    \
    found:                                             \
    value;                                             \
})
```

Labels As Values

You can get the address of a label defined in a current function using the operator && (a unary operator, for you language lawyers). The value has type void *. The label address is a constant value—it can be used wherever a constant of that type is valid:

```
void *ptr;
ptr = &&foo;
```

To use this value, you need to be able to jump to it. With a normal label, you might say goto ptr;. However, with a label used as a value, you use a computed goto statement:

```
goto *ptr;
```

Any expression that evaluates to the type void * is allowed. One way to use these computed gotos is to initialize static arrays that will serve as a jump table:

```
static void *jump[] = { &&f, &&g, &&h };
```

In this case, f, g, and h are labels to which code jumps at runtime. To use one, you index into the array and select the label. The code to do this might resemble the following:

```
goto *array[i];
```

You cannot use computed gotos to jump to code in a different function, primarily because attempting to do so subverts (or, perhaps, abuses) the local label feature described in the previous section. Store the label addresses in automatic variables and never pass such an address as an argument to another function.

Nested Functions

A *nested function* is a function defined inside another function. As you might anticipate, a nested function's name is local to the block in which it is defined. To illustrate, consider a nested function named `swap()` that is called twice:

```
f(int i, int j)
{
        void swap(int *a, int *b)
        {
                int tmp = *a;
                *a = *b;
                *b = tmp;
        }
/* more code here */
        swap(&i, &j);
}
```

The nested `swap()` can access all the variables of the surrounding function that are visible at the point when it is defined (known as lexical scoping).

■ **Note** GNU C++ does not support nested functions.

Nested function definitions are permitted in the same places in which variable definitions are allowed within functions: in any block and before the first statement in the block.

It is possible to call the nested function from outside the scope of its name by storing its address or passing the address to another function:

```
inc(int *array, int size)
{
        void swap(int *a, int *b)
        {
                int tmp = *a;
                *a = *b;
                *b = tmp;
        }
        save(swap, size);
}
```

In this snippet, the address of `swap()` is passed as an argument to the function `save()`. This trick only works if the surrounding function, `inc()`, does not exit.

If you try to call a nested function through its address after the surrounding function exits, well, as GCC authors put it, "all hell will break loose." The most common disaster is that variables that have gone out of scope will no longer exist and you will end up referencing garbage or overwriting something else, either of which can be a difficult bug to track down. You might get lucky, but the path of true wisdom is not to take the risk.

■ **Note** GCC uses a technique called trampolines to implement taking addresses of nested functions. You can read more about this technique at `http://gcc.gnu.org/onlinedocs/gccint/Trampolines.html`.

Nested functions can jump to a label inherited from a containing function, that is, a local label, if the label is explicitly declared in the containing function, as described in the section "Locally Declared Labels."

A nested function always has internal linkage, so declaring a nested function with `extern` is an error.

Constructing Function Calls

Using GCC's built-in functions, you can perform some interesting tricks, such as recording the arguments a function receives and calling another function with the same arguments, but without knowing in advance how many arguments were passed or the types of the arguments.

```
void * __builtin_apply_args();
```

`__builtin_apply_args()` returns a pointer to data describing how to perform a call with the same arguments as were passed to the current function.

```
void * __builtin_apply(void (*function)(), void *args, size_t size);
```

`__builtin_apply()` calls `function` with a copy of the parameters described by `args` and `size`. `args` should be the value returned by `__builtin_apply_args()`. The argument size specifies the size of the stack argument data in bytes. As you can probably imagine, computing the proper value for size can be nontrivial, but can usually be done by simply calculating the sum of the `sizeof()` each of the parameters. For more complex data structures, it is often easiest to simply specify a sufficiently large number for size, but this will waste space and is somewhat inelegant.

Similarly, and again without advance knowledge of a function's return type, you can record that function's return value and return it yourself. Naturally, though, the calling function must be prepared to receive that datatype. The built-in function that accomplishes this feat is `__builtin_return()`.

```
void __builtin_return(void *retval);
```

`__builtin_return()` returns the value described by `retval` from the containing function. `retval` must be a value returned by `__builtin_apply()`.

```
#include <stdio.h>

int function1(char * string, int number) {

  printf("funtion1: %s\n", string);
  return number + 1;

}

int function2(char * string, int number) {
  void* arg_list;
  void* return_value;

  arg_list = __builtin_apply_args();
  return_value =  __builtin_apply( (void*) function1, arg_list, ~CCC
    sizeof(char *) + sizeof(int));
  __builtin_return(return_value);

}
```

```
int main(int argc, char ** argv) {

  printf("returned value: %d\n", function2("hello there", 42));

  return 0;
}
```

Referring to a Type with typeof

Another way to refer to or obtain the type of an expression is to use the `typeof` keyword, which shares its syntax with the ISO C keyword `sizeof` but uses the semantics of a type name defined with `typedef`. Syntactically, `typeof`'s usage is what you would expect: you can use `typeof` with an expression or with a type. With an expression, the usage might resemble the following:

```
typeof (array[0](1));
```

As you can see in this example, `array` is an array of pointers to functions. The resulting type will be the values of the functions. `typeof`'s usage with a type name is more straightforward.

```
typeof (char *);
```

Obviously, the resulting type will be pointers to `char`.

■ **Tip** If you are writing a header file that must work when included in ISO C programs, write `__typeof__` instead of `typeof`.

Semantically, `typeof` can be used anywhere a `typedef` name would be, which includes declarations, casts, `sizeof` statements, or even, if you want to be perverse, within another `typeof` statement. A typical use of `typeof` is to create type-safe expressions, such as a `max` macro that can safely operate on any arithmetic type and evaluate each of its arguments exactly once:

```
#define max(a,b)         \
 ({ typeof (a) _a = (a); \
    typeof (b) _b = (b); \
    _a > _b ? _a : _b; })
```

The names `_a` and `_b` start with underscores to avoid the local variables conflicting with variables having the same name at the scope in which the macro is called. Additional `typeof` uses might include the following, or variations thereof:

- `typeof (*x) y;`: Declares `y` to be of the type to which `x` points.
- `typeof (*x) y[4];`: Declares `y` to be an array of the values to which `x` points.
- `typeof (typeof (char *)[4]) y;`: Declares `y` as an array of pointers to characters. Clearly, this code is more elaborate than the standard C declaration `char *y[4];`, to which it is equivalent.

Zero-Length Arrays

Zero-length arrays are allowed in GNU C. They are very useful as the last element of a structure that is really a header for a variable-length object. Consider the following:

```
struct data {
        int i;
        int j;
}

struct entry {
        int size;
        struct data[0];
};

int e_sz;
struct entry *e = (struct entry *) malloc(sizeof (struct entry) + e_sz);
e->size = e_sz;
```

In ISO C89, you must define `data[1]`, giving `data` a length of 1. This requirement forces you to waste space (a trivial concern in this example) or, more likely, complicate the `malloc()` call. ISO C99 allows you to use flexible array members, which have slightly different characteristics:

- Flexible array members would be defined as `contents[]`, not `contents[0]`.
- C99 considers flexible array members to have incomplete type specifications. You cannot use `sizeof` with incomplete types. With GNU C's zero-length arrays, however, `sizeof` evaluates to 0. It is up to you to decide if that is a feature or not.
- Most significantly, flexible array members may only appear as the last member of nonempty structs.

To support flexible array members, GCC extends them to permit static initialization of flexible array members. This is equivalent to defining two structures where the second structure contains the first one, followed by an array of sufficient size to contain the data. If that seems confusing, consider the two structures `s1` and `s2` defined as follows:

```
struct f1 {
        int x;
        int y[];
} f1 = { 1, { 2, 3, 4 } };

struct f2 {
        struct f1 f1;
        int data[3];
} f2 = { { 1 }, { 2, 3, 4 } };

struct s1 {
        int i;
        int j[];
} s1 = {1, {2, 3, 4} };

struct s2 {
        struct s1 s1;
        int array[3];
} s2 = { {1}, {2, 3, 4} };
```

In effect, `s1` is defined as if it were declared as `s2`.The convenience of this extension is that `f1` has the desired type, eliminating the need to consistently refer to the awkward construction `f2.f1`. This extension is also symmetric with declarations of normal static arrays, in that an array of unknown size is also written with `[]`. For example, consider the following code:

```
struct foo {
        int x;
        int y[];
};
struct bar {
        struct foo z;
};

struct foo a = {1, {2, 3, 4 } };          /* valid   */
struct bar b = { { 1, {2, 3, 4 } } };     /* invalid */
struct bar c = { { 1, {} } };             /* valid   */
struct foo d[1] = { { 1 {2, 3, 4} } };    /* invalid */
```

■ **Note** This code will not compile unless you are using GCC 3.2 or newer.

Arrays of Variable Length

ISO C99 allows variable-length automatic arrays, which enable you to have a nonconstant expression for the array size but also have arrays that are preallocated at a fixed size (unlike variable arrays in Java). Not wanting to limit this very handy feature to C99, GCC accepts variable-length arrays in C89 mode and in C++ (using `-std=gcc89`). Nevertheless, what ISO C99 giveth, GCC taketh away: GCC's implementation of variable-length arrays does not yet conform in all details to the ISO C99 standard. Variable-length arrays are declared like other automatic arrays, but the length is not a constant expression. The storage is allocated at the point of declaration and deallocated when the brace level is exited. For example

```
FILE *myfopen(char *s1, char *s2, char *mode)
{
        char str[strlen (s1) + strlen (s2) + 1];
        strcpy(str, s1);
        strcat(str, s2);
        return fopen(str, mode);
}
```

Jumping or breaking out of the scope of the array name deallocates the storage. You can use the function `alloca()` to get an effect much like variable-length arrays. The function `alloca()` is available in many, but not all, C implementations. On the other hand, variable-length arrays are more elegant. You might find variable-length arrays more straightforward to use because the syntax is more natural.

Should you use an `alloca()`-declared array or a variable-length array? Before you decide, consider the differences between these two methods. Space allocated with `alloca()` exists until the containing function returns, whereas space for a variable-length array is deallocated as soon as the array name's scope ends. If you use both variable-length arrays and `alloca()` in the same function, deallocation of a variable-length array will also deallocate anything more recently allocated with `alloca()`. You can also use variable-length arrays as arguments to functions.

```
void f(int len, char data[len])
{
        /* function body here */
}
```

In this example, you can see the function `foo()` accepts the integer parameter `len`, which is also used to specify the size of the array of `data`. The array's length is determined when the storage is allocated and is remembered for the scope of the array, in case the length is accessed using `sizeof`.

If you want to pass the array first and the length second, you can use a forward declaration in the parameter list (which is, by the way, another GNU extension). A forward declaration in the parameter list looks like the following:

```
void f(int len; char data[len], int len)
{
        /* function body here */
}
```

The declaration `int len` before the semicolon, known as a *parameter forward declaration*, makes the name `len` known when the declaration of `data` is parsed.

You can make multiple parameter forward declarations in the parameter list, separated by commas or semicolons, but the last one must end with a semicolon. Following the final semicolon, you must declare the actual parameters. Each forward declaration must match a real declaration in parameter name and datatype. ISO C99 does not support parameter forward declarations.

Macros with a Variable Number of Arguments

C99 allows declaring macros with a variable number of arguments, just as functions can accept a variable number of arguments. The syntax for defining the macro is similar to that of a function. For example:

```
#define debug(format, ...) fprintf(stderr, format, __VA_ARGS__)
```

The ellipsis specifies the variable argument. When you invoke such a macro, the ellipsis represents zero or more tokens until the closing parenthesis that ends the invocation. This set of tokens replaces the identifier `__VA_ARGS__` in the macro body wherever it appears. See the C preprocessor manual (info cpp) for more information about how GCC processes variadic arguments.

Subscripting Non-lvalue Arrays

A common C language term, lvalues, is a reference to objects that are addressable and can be examined, but may or may not be assignable. In ISO C99, arrays that are not lvalues still decay to pointers and can therefore be subscripted, though they cannot be modified or used after the next sequence point, and the unary & operator cannot be applied to them. (A sequence point is a point in a program's execution where all statements whose order of evaluation is potentially ambiguous have been resolved. The simplest example of a sequence point is the semicolon that terminates any combination of C statements.) Non-lvalue arrays are therefore arrays whose existence has been declared but to which explicit values have not been assigned.

As an extension, GCC's C compiler allows non-lvalue arrays to be subscripted in C89 mode. For example, the following code is valid in GNU C but not in C89:

```
#include <stdio.h>

struct foo {
        int a[4];
};

struct foo f();

bar(int index)
{
        return f().a[index];
}
```

Being able to reference arrays whose elements may not have values simplifies coding, but can introduce subtle, data-dependent bugs in your code.

Arithmetic on Void and Function Pointers

GNU C supports adding and subtracting pointer values on pointers to void and on pointers to functions. GCC implements this feature by assuming that the size of a void or a function is 1. As a consequence, `sizeof` also works on void and function types, returning 1. The option `-Wpointer-arith` enables warnings if these extensions are used.

Nonconstant Initializers

GNU C does not require the elements of an aggregate initializer for automatic variables to be constant expressions. This is the same behavior permitted by both standard C++ and ISO C99. An initializer with elements that vary at runtime might resemble the following code:

```
float f(float f, float g)
{
        float beats [2] = { f-g, f+g };
        /* function body here */
        return beats[1];
}
```

Designated Initializers

Standard C89 requires initializer elements to appear in the same order as the elements in the array or structure being initialized. C99 relaxes this restriction, permitting you to specify the initializer elements in any order by specifying the array indices or structure field names to which the initializers apply. These are known as *designated initializers*. GNU C allows this as an extension in C89 mode, but not—for you C++ programmers—in GNU C++.

To specify an array index, for example, write `[`*`index`*`] =` before the element value as shown here:

```
int a[6] = {[4] = 29, [2] = 15};
```

This is equivalent to

```
int a[6] = {0, 0, 15, 0, 29, 0};
```

The index values must be a constant expression, even if the array being initialized is automatic.

To initialize a range of elements with the same value, use the syntax `[`*`first ... last`*`]=` *`value`*. This is a GNU extension. For example

```
int widths[] = {[0 ... 9] = 1, [10 ... 99] = 2, [100] = 3};
```

If *value* has side effects, the side effects happen only once, rather than for each element initialized.

In a structure initializer, specify the name of a field to initialize with `.member=`. Given a structure triplet that is defined as

```
struct triplet {
        int x, y, z;
};
```

The following code snippet illustrates the proper initialization method:

```
struct triplet p = {
        .y = y_val,
        .x = x_val,
        .z = z_val
};
```

This initialization is equivalent to

```
struct triplet p = {
        x_val,
        y_val,
        z_val
};
```

The [*index*] or .member is referred to as a designator.

You can use a designator when initializing a union in order to identify the union element you want to initialize. For example

```
union foo {
        int i;
        double d;
};
```

The following statement converts 4 to a double to store it in the union using the second element:

```
union foo f = {
        .d = 4
};
```

In contrast, casting 4 to type `union foo` would store it in the union as the integer i, because it is an integer.

You can combine designated initializers with ordinary C initialization of successive elements. In this case, initializers lacking designators apply to the next consecutive element of the array or structure. For example, the line

```
int a[6] = { [1] = v1, v2, [4] = v4 };
```

is equivalent to

```
int a[6] = { 0, v1, v2, 0, v4, 0 };
```

Labeling the elements of an array initializer is especially useful when the indices are characters or belong to an `enum` type. For example

```
int whitespace[256] = {
        [' '] = 1, ['\t'] = 1, ['\v'] = 1,
        ['\f'] = 1, ['\n'] = 1, ['\r'] = 1 };
```

You can also write a series of .member and [*index*] designators before = to specify a nested subobject to initialize; the list is taken relative to the subobject corresponding to the closest surrounding brace pair. For example, with the preceding `struct triplet` declaration, you would use the following:

```
struct triplet triplet_array[10] = { [2].y = yv2, [2].x = xv2, [0].x = xv0 };
```

■ **Note** If the same field is initialized multiple times, its value will be from the last initialization. If multiple overridden initializations have side effects, the standard does not specify whether the side effect will happen or not. GCC currently discards additional side effects and issues a warning.

Case Ranges

You can specify a range of consecutive values in a single case label like this:

```
case m ... n:
```

Spaces around the ellipsis (...) are required. This has the same effect as the proper number of individual case labels, one for each integer value from `m` to `n`, inclusive. This feature is especially useful for ranges of ASCII character codes and numeric values, as in the following example:

```
case 'A' ... 'Z':
```

Mixed Declarations and Code

ISO C99 and ISO C++ allow declarations and code to be freely mixed within compound statements. The GCC extension to this feature allows mixed declarations and code in C89 mode. For example, you can write

```
int i;
/* other declarations here */
i++;
int j = i + 2;
```

Each identifier is visible from the point at which it is declared until the end of the enclosing block.

Declaring Function Attributes

GNU C enables you to tell the compiler about certain features or behaviors of the functions in your program. This is done using keywords that declare *function attributes*. Function attributes permit the compiler to optimize function calls and check your code more carefully. The keyword `__attribute__` allows you to specify function attributes. `__attribute__` must be followed by an attribute specification inside double parentheses. The following attributes are currently defined for functions on all targets:

- `alias`
- `always_inline`
- `cdecl`
- `const`
- `constructor`
- `deprecated`
- `destructor`
- `dllexport`
- `dllimport`
- `fastcall`

- flatten
- format
- format_arg
- malloc
- no_instrument_function
- noinline
- nonull
- noreturn
- pure
- regparm
- section
- stdcall
- unused
- used
- warn_unused_result
- weak

These attributes are explained throughout the remainder of this section. This list includes x86-specific attributes, since x86 is the most popular GCC platform. Several other attributes are defined for functions on particular target systems—these are discussed in Appendix B of this book, which discusses platform-specific options. Other attributes are supported for variable declarations and for types.

■ **Tip** You can also specify attributes by embedding each keyword between double underscores (__). This alternative syntax allows you to use function attributes in header files without being concerned about a possible macro of the same name. For example, you can write `__noreturn__` instead of `__attribute__((noreturn))`.

The `alias` attribute enables you to identify a declaration as an alias for an existing, predefined symbol elsewhere in your code.

Generally, functions are not inline unless optimization is specified. For functions declared inline, the `always_inline` attribute inlines the associated function even in the absence of optimization. Conversely, to prevent a function from ever being considered for inlining, use the function attribute `noinline`.

On x86 systems, the `cdecl` attribute causes the compiler to assume that the calling function will clean up the stack, removing the storage used to pass arguments. This is useful in conjunction with the `-mrtd` switch, which causes calling functions to use the `ret` *NUM* convention that pops arguments off the stack during the return to the calling function.

Many functions do not examine or modify any values except their arguments or have any effect other than returning a value. Such functions can be given the `const` attribute. A function that has pointer arguments and examines the data pointed to must not be declared `const`. Likewise, a function that calls a non-`const` function cannot be `const`. Naturally, it does not make sense for a `const` function to return `void`.

The `deprecated` attribute results in a warning if the associated function is used anywhere in the source file. This attribute is useful to identify functions you expect to remove in a future version of a

program. The warning also includes the location of the declaration of the deprecated function, enabling users to find further information about why the function is deprecated, or what they should do instead. The warning only occurs when the function's return value is used. For example

```
int old_f() __attribute__ ((deprecated));
int old_f();
int (*f_ptr)() = old_f;
```

The third line generates a warning, but the second line does not.

■ **Tip** You can also use the `deprecated` attribute with variables and `typedef`s.

The `ddlexport` and `dllimport` attributes are only useful when using GCC for Microsoft Windows or Symbian OS targets, and respectively provide a global pointer to a function and enable access to a function through a global pointer.

On x86 systems and for functions that pass a fixed number of arguments, the `fastcall` attribute tells the compiler to pass the first two arguments in the registers ECX and EDX as an optimization. Any other arguments are passed on the stack, and are popped by the function that you are calling. If you are calling a function with a variable number of arguments this attribute is ignored and all arguments are pushed on the stack.

The `flatten` attribute tells gcc to inline every instance of a call to a specific function whenever possible, though whether inlining actually occurs is still dependent on the inlining options specified on the command line. Like the `always_inline` attribute, this attribute provides an interesting alternative to defining macros when you only want certain functions to be inline whenever possible.

The `malloc` attribute tells the compiler that a function should be treated as if it were the `malloc()` function. The purpose for this attribute is that the compiler assumes that calls to `malloc()` result in a pointer that cannot alias anything. This will often improve optimization specifically during alias analysis (see Chapter 5 for discussion of GCC alias analysis during optimization).

The `nonnull (arg-index, ...)` attribute is useful to check that function parameters are `nonnull` pointers, and takes a list of such arguments. Using this attribute generates a warning if any of the specified arguments are actually `NULL`. If no parenthesized argument list is supplied, all parameters are verified not to be `NULL`.

A few standard library functions, such as `abort()` and `exit()`, cannot return (they never return to the calling function). GCC automatically knows this about standard library functions and its own built-in functions. If your own code defines functions that never return, you can declare them using the `noreturn` attribute to tell the compiler this fact. For example

```
void bye(int error) __attribute__ ((noreturn));

void bye(int error)
{
        /* error handling here*/
        exit(1);
}
```

The `noreturn` keyword tells the compiler to assume that `bye()` cannot return. The compiler can then optimize without needing to consider what might happen if `bye()` does return, which can result in slightly better code. Declaring functions `noreturn` also helps avoid spurious warnings about uninitialized variables. However, you should not assume that registers saved by the calling function are restored before calling the `noreturn` function. If a function does not return and is given the `noreturn` attribute, it should not have a return type other than `void`.

Many functions have no effects to return a value. Similarly, such functions' return values often depend only on the function parameters and/or global variables. As you will learn in Chapter 5, such functions can easily be optimized during common subexpression elimination and loop optimization, just as arithmetic operators would be. Such functions should be declared with the `pure` attribute.

For example, the following function declaration asserts that the `cube()` function is safe to optimize using common subexpression elimination:

```
int cube(int i) __attribute__ ((pure));
```

Functions that might benefit from declaration as `pure` functions include functions that resemble `strlen()` and `memcmp()`. Functions you might not want to declare using the `pure` attribute include functions with infinite loops and those that depend on volatile memory or other system resources. The issue with such functions is that they depend on values that might change between two consecutive calls, such as `feof()` in a multithreading environment.

Similar to the `fastcall` attribute, the `regparm (`*`number`*`)` attribute causes the compiler to pass up to number integer arguments in registers EAX, EDX, and ECX instead of on the stack. Like `fastcall`, this only applies to functions that take a fixed number of arguments—functions that take a variable number of arguments will continue to be passed all of their arguments on the stack.

On x86 systems, the `stdcall` attribute causes the compiler to assume that the function that is being called will clean up the stack space used to pass arguments.

The function attribute `unused` means that it might not be used and that this is acceptable. Accordingly, GCC will omit producing a warning for this function. Likewise, the `used` function attribute declares that code must be emitted for the function even if it appears that the function is never referenced. This is useful, for example, when the function is referenced only in inline assembly.

The `warn_unused_result` attribute causes a warning to be emitted if a caller of the function with this attribute does not use its return value. This can be extremely useful in detecting potential security problems or a definite bug, such as with the `realloc` function.

■ **Note** GNU C++ does not currently support the `unused` attribute because definitions without parameters are valid in C++.

Before the language lawyers among you start complaining that ISO C's pragma feature should be used instead of `__attribute__`, consider the following points the GCC developers make in the GCC Texinfo help file:

> *At the time `__attribute__` was designed, there were two reasons for not using #pragma:*
>
> *1. It is impossible to generate #pragma commands from a macro.*
>
> *2. There is no telling what the same #pragma might mean in another compiler.*

These two reasons applied to almost any application that might have been proposed for #pragma. It was basically a mistake to use #pragma for anything.

The first point is somewhat less relevant now, because the ISO C99 standard includes `_Pragma`, which allows pragmas to be generated from macros. GCC-specific pragmas (`#pragma GCC`), moreover, now have their own namespace. So why does `__attribute__` persist? Again, GCC developers explain that "it has been found convenient to use `__attribute__` to achieve a natural attachment of attributes to their corresponding declarations, whereas `#pragma GCC` is of use for constructs that do not naturally form part of the grammar."

Specifying Variable Attributes

You can also apply `__attribute__` to variables. The syntax is the same as for function attributes. GCC supports ten variable attributes:

- `aligned`
- `deprecated`
- `mode`
- `nocommon`
- `packed`
- `section`
- `transparent_union`
- `unused`
- `vector_size`
- `weak`

To specify multiple attributes, separate them with commas within the double parentheses: for example, `__attribute__ ((aligned (16),packed))`. GCC defines other attributes for variables on particular target systems. Other front ends might define more or alternative attributes. For details, consult the GCC online help (info gcc).

The aligned (`n`) attribute specifies a minimum alignment of `n` bytes for a variable or structure field. For example, the following declaration tells the compiler to allocate a global variable `j` aligned on a 16-byte boundary:

```
int j __attribute__ ((aligned (16))) = 0;
```

You can also specify the alignment of structure fields. For example, you can create a pair of `int`s aligned on an 8-byte boundary with the following declaration:

```
struct pair {
        int x[2] __attribute__ ((aligned (8)));
};
```

If you choose, you can omit a specific alignment value and simply ask the compiler to align a variable or a field in a way that is appropriate for the target.

```
char s[3] __attribute__ ((aligned));
```

An `aligned` attribute lacking an alignment boundary causes the compiler to set the alignment automatically to the largest alignment ever used for any datatype on the target machine. Alignment is a valuable optimization because it can often make copy operations more efficient. How? The compiler can use native CPU instructions to copy natural memory sizes when performing copies to or from aligned variables or fields.

■ **Tip** The `aligned` attribute increases alignment; to decrease it, specify `packed` as well.

The `deprecated` attribute has the same effect and behavior for variables as it does for functions.

The `mode` (`m`) attribute specifies the datatype for the declaration with a type corresponding to the mode `m`. In effect, you declare an integer or floating-point type by width rather than type. Similarly,

you can specify a mode of `byte` or `__byte__` to declare a mode of a one-byte integer; `word` or `__word__` for a one-word integer mode, or `pointer` or `__pointer__` for the mode used to represent pointers.

The `packed` attribute requests allocating a variable or struct member with the smallest possible alignment, which is 1 byte for variables and 1 bit for a field. You can specify a larger alignment with the `aligned` attribute. The following code snippet illustrates a struct in which the field `s` is packed, which means that `s` immediately follows `index`—that is, there is no padding to a natural memory boundary.

```
struct node
{
        int index;
        char s[2] __attribute__ ((packed));
        struct node *next;
};
```

Normally, the compiler places the code objects in named sections such as `data` and `bss`. If you need additional sections or want certain particular variables to appear in special sections, you can use the attribute `section (name)` to obtain that result. For example, an operating system kernel might use the section name `.kern_data` to store kernel-specific data. The `section` attribute declares that a variable or a function should be placed in a particular section.

The following small program, adapted from the GCC documentation, declares several section names, `DUART_A`, `DUART_B`, `STACK`, and `INITDATA`:

```
struct duart a __attribute__ ((section ("DUART_A"))) = { 0 };
struct duart b __attribute__ ((section ("DUART_B"))) = { 0 };
char stack[10000] __attribute__ ((section ("STACK"))) = { 0 };
int init_data __attribute__ ((section ("INITDATA"))) = 0;

int main(void)
{
        /* set up stack pointer */
        init_sp(stack + sizeof (stack));

        /* set up initialized data */
        memcpy(&init_data, &data, &edata - &data);

        /* enable the serial ports */
        init_duart(&a);
        init_duart(&b);
}
```

Note that the sample program uses the `section` attribute with an initialized definition of a global variable, such as the four global definitions at the beginning of the program. If you fail to use an initialized global variable, GCC will emit a warning and ignore the `section` attribute applied to uninitialized variable declarations. This restriction exists because the linker requires each object be defined once. Uninitialized variables are temporarily placed in the `.common` (or `.bss`) section and so can be multiply-defined. To force a variable to be initialized, specify the `-fno-common` flag or declare the variable using the `nocommon` attribute.

■ **Tip** Some executable file formats do not support arbitrary sections; so the `section` attribute is not available on those platforms. On such platforms, use the linker to map the contents of a module to a specific section.

The `transparent_union` attribute is used for function parameters that are unions. A transparent union declares that the corresponding argument might have the type of any union member, but that the argument is passed as if its type were that of the first union member. The `unused` attribute has the same syntax and behavior as the `unused` attribute for functions.

Inline Functions

When you use `inline` with a function, GCC attempts to integrate that function's code into its callers. As an optimization, inline functions make execution faster by eliminating the overhead of function calls (saving and restoring stack pointers, for example). If the argument values are constant, they can be optimized at compile time, reducing the amount of function code integrated into the callers. Although at first sight inlining might seem to inflate code size, this is not necessarily the case. Other optimizations might allow sufficient code hoisting or subexpression elimination in such a way that the actual integrated code is smaller.

Although C99 includes inline functions, one of the shortcomings of GCC is that its implementation of inline functions differs from requirements of the standard. To declare a function inline, use the `inline` keyword in its declaration:

```
inline long cube(long i)
{
        return i * i * i;
}
```

■ **Note** If you are writing a header file for inclusion in ISO C programs, use `__inline__` instead of `inline`.

In the absence of the `inline` attribute, or in addition to it, you can instruct GCC to inline all "simple enough" functions by specifying the command-line option `-finline-functions`, where GCC decides what constitutes a "simple enough" function.

Code constructs that make it impossible (or, at the least, extremely difficult) to inline functions include using variadic arguments; calling `alloca()` in the function body; using variable-sized datatypes (such as variable-length arrays); jumping to computed and nonlocal `goto`s; calling functions before their definition; including recursive function calls with a function definition; and nesting functions. If there is a nonintegrated call, then the function is compiled to assembler code as usual. The function must also be compiled as usual if the program refers to its address, because that cannot be inlined.

If you need to know when requested inlines can not be implemented, specify the command-line option `-Winline` to emit both a warning that a function could not be inlined and an explanation of why it could not be inlined.

For functions that are both inline and static, that function's assembly language code will never be referenced if, first, all calls to that function are integrated into a calling function, and, second, the integrated function's address is never taken. Accordingly, GCC will not even emit assembler code for the function, which you can override by specifying the option `-fkeep-inline-functions`.

Anticipating future compatibility with C99 semantics for inline functions, GCC's developers recommend using only static inline for inline functions. Why? The existing semantics will continue to function as expected when `-std=gnu89` is specified, but the eventual default for inline will be GCC's behavior when `-std=gnu99` is specified. The issue is that `-std=gnu99` will implement the C99 semantics but that it does not yet do so.

■ **Tip** GCC does not inline any functions when not optimizing unless you specify the `always_inline` attribute for the function, such as

```
inline void f (const char) __attribute__ ((always_inline));
```

Function Names As Strings

GCC predefines two magic identifiers that store the name of the current function. `__FUNCTION__` stores the function name as it appears in the source code; `__PRETTY_FUNCTION__` stores the name *pretty* printed in a language-specific fashion. In C programs, the two function names are the same, but in C++ programs, they will probably be different. Consider the following source code from the file FUNCTION_example.c:

```
#include <stdio.h>

void here(void)
{
  printf("Function %s in %s\n",  __FUNCTION__, __FILE__);
  printf("Pretty Function %s in %s\n",  __PRETTY_FUNCTION__, __FILE__);
}

int main(void)
{
        here();
        return 0;
}
```

Running the resulting program, you get the following:

```
$ ./a.out
Function here in FUNCTION_example.c
Pretty Function here in FUNCTION_example.c
```

Because `__FUNCTION__` and `__PRETTY_FUNCTION__` are not macros, `#ifdef __FUNCTION__` is meaningless inside a function because the preprocessor does not do anything special with the identifier `__FUNCTION__` (or `__PRETTY_FUNCTION__`).

A third related, predefined identifier is `__LINE__`, which holds the current line number in the source code. As an example, consider the following slightly modified source code from the source file `FUNCTION_example_with_line.c`:

```
#include <stdio.h>

void here(void)
{
  printf("Function %s in %s, line %d\n",  __FUNCTION__, __FILE__, __LINE__);
  printf("Pretty Function %s in %s, line %d\n",  __PRETTY_FUNCTION__, \
         __FILE__, __LINE__);
}
```

```
int main(void)
{
        here();
        return 0;
}
```

Running the resulting program, you get the following:

```
$ ./a.out
Function here in FUNCTION_example_with_line.c, line 5
Pretty Function here in FUNCTION_example_with_line.c, line 6
```

#pragmas Accepted by GCC

GCC supports several types of #pragmas, primarily in order to compile code originally written for other compilers. Pragmas essentially enable you to embed special instructions when code is compiled on specific platforms or under specific circumstances—you can think of them as a special case of platform-specific `#ifdef`. Note that in general I do not recommend the use of pragmas. In particular, GCC defines pragmas for ARM, Darwin, Solaris, and Tru64 systems.

ARM #pragmas

ARM targets define #pragmas for controlling the default addition of `long_call` and `short_call` attributes to functions.

- `long_calls`: Enables the `long_call` attribute for all subsequent functions.
- `no_long_calls`: Enables the `short_call` attribute for all subsequent functions.
- `long_calls_off`: Disables the `long_call` and `short_call` attributes for all subsequent functions.

Darwin #pragmas

The following #pragmas are available for all architectures running the Darwin operating system. These are useful for compatibility with other Mac OS compilers.

- `mark token`: Is accepted for compatibility, but otherwise ignored.
- `options align=target`: Sets the alignment of structure members. The values of target may be `mac68k`, to emulate m68k alignment, or `power`, to emulate PowerPC alignment. `reset` restores the previous setting.
- `segment token`: Is accepted for compatibility, but otherwise ignored.
- `unused (var [, var]...)`: Declares variables as potentially unused, similar to the effect of the attribute `unused`. Unlike the `unused` attribute, #pragma can appear anywhere in variable scopes.

Solaris #pragmas

For compatibility with the SunPRO compiler, GCC's C compiler supports the `redefine_extname oldname newname` pragma. This #pragma assigns the assembler label `newname` to the C function `oldname`, which is equivalent to the `asm` label's extension. The #pragma must appear before the function declaration. The preprocessor defines `__PRAGMA__REDEFINE_EXTNAME` if this #pragma is available.

Tru64 #pragmas

GCC's C compiler supports the `extern_prefix` *`string`* #pragma for compatibility with the Compaq C compiler. #pragma `extern_prefix` *`string`* prefixes the value of *string* to all subsequent function and variable declarations. To terminate the effect, use another #pragma `extern_prefix` with an empty string. The preprocessor defines `__PRAGMA_EXTERN_PREFIX` if this #pragma is available.

Objective-C Support in GCC's C Compiler

Objective-C is an object-oriented superset of C with extensions that provide a message-passing interface similar to that provided by the Smalltalk-80 language. Objective-C was originally written by Brad J. Cox and Kurt J. Schmucker at Stepstone Corporation (originally known as Productivity Products International). Objective-C's primary goal is to add some of the promises of truly reusable code to the C language by adding some of the core concepts of Smalltalk-80 while leaving behind baggage such as the fact that most Smalltalk environments ran on virtual machines rather than as standalone compiled code. Unfortunately, the virtual machine concept lives on, to some extent, in Objective-C's heavy reliance on a runtime environment, which means that it is not possible to compile standalone (i.e., static) Objective-C code.

Though excellent in both concept and implementation, Objective-C was saved from potential obscurity by its adoption in 1988 as the default language used for developing NeXTSTEP applications. NeXTSTEP was the operating system and execution environment used on Steve Jobs' NeXT computers, and is now the parent of the Cocoa execution and development environment used on Mac OS X. Objective-C support was added to GCC's C compiler, gcc, beginning in 1992, and is heavily used in the GNUstep project.

The fact that Objective-C is a pure superset of C differentiates it from C++, while its dynamic typing and other runtime features distinguish it from the other object-oriented languages supported by GCC, namely C++ and Java. The Objective-C runtime library supports accessing methods and classes by their string names, does a significant amount of typing at runtime (i.e., when you actually execute your application), and supports the addition of classes and categories at runtime. There are actually two standard Objective-C runtime libraries/environments available. If you are running applications on Mac OS X platforms, you will link against the NeXT runtime library that is present on that platform—on all other platforms, you will compile and link with gcc's default Objective-C library. Regardless of whether you build gcc yourself for the Mac OS X platform or install a precompiled version as part of Apple's Xcode tools development environment, its default behavior on OS X systems is to expect and link with the NeXTSTEP/OS X Objective-C runtime.

IDENTIFYING LIBRARY DEPENDENCIES

Given the differences in capabilities between the GNU and the NeXTSTEP runtimes, it is often useful to identify the libraries that your application has linked against. This is especially important if you have built your own version of the Objective-C compiler for your Mac OS X system. By default, Apple's gcc compiler (provided with its Xcode development environment) includes and uses the NeXTSTEP Objective-C runtime. If you've built your own gcc, you may be using its runtime and therefore cannot use all of the capabilities provided by the NeXTSTEP runtime.

The traditional mechanism for listing library dependencies used by the loader to resolve symbols is to use the ldd program, which is traditionally built and installed as part of the GNU C library, Glibc. On Linux systems and most other systems with Glibc installed, its output looks something like the following for a traditional Objective-C application:

```
$ ldd hello
 libobjc.so.2 => /usr/local/gcc4.1svn/lib64/libobjc.so.2 (0x00002aaaaabc2000)
 libgcc_s.so.1 => /usr/local/gcc4.1svn/lib64/libgcc_s.so.1 (0x00002aaaaacde000)
 libc.so.6 => /lib64/tls/libc.so.6 (0x00002aaaaadec000)
 /lib64/ld-linux-x86-64.so.2 (0x00002aaaaaaab000)
```

Okay, maybe that's not so standard—you can see that I'm using a 64-bit system and running my own version of gcc, built from the latest Subversion sources. Well, you wanted this book to be up-to-date, right? Regardless, it at least tells me what libraries my application depends on.

Unfortunately, the ldd application is not provided by default as part of the Xcode gcc environment on Mac OS X systems. To list library dependencies on OS X systems, you'll need to use the `otool -L` command, as in the following example:

```
$ otool -L hello
hello:
/usr/lib/libobjc.A.dylib (compatibility version 1.0.0, current version 227.0.0)
/usr/lib/libgcc_s.1.dylib (compatibility version 1.0.0, current version 1.0.0)
/usr/lib/libmx.A.dylib (compatibility version 1.0.0, current version 92.0.0)
/usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 88.1.2)
```

This example shows that this application uses the standard libraries provided with the default gcc delivered with Xcode. However, after compiling my application with a hand-built version of gcc for OS X, I get the following output:

```
$ otool -L hello
hello:
/usr/lib/libobjc.A.dylib (compatibility version 1.0.0, current version 227.0.0)
/usr/local/lib/libgcc_s.1.0.dylib (compatibility version 1.0.0, ➥
  current version 1.0.0)
/usr/lib/libmx.A.dylib (compatibility version 1.0.0, current version 92.0.0)
/usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 88.1.2)
```

In this case, my application is still using the default NeXTSTEP runtime, but is using its own version of libgcc, which should work correctly in the OS X environment. However, if you encounter problems or "new behavior" after recompiling an existing application with a hand-built version of gcc on Mac OS X systems, it's worth quickly checking library dependencies before plunging into an orgy of `printf()`'s and other more classic diagnostic techniques.

Like the other chapters in this book that explain how to use the GCC compilers for different languages, this section is not a tutorial on the Objective-C programming language. Some excellent sites that provide that sort of information include the following:

- *Objective-C Links, Resources, Stuff*: `http://www.foldr.org/~michaelw/objective-c/`
- *About Objective-C*: `http://www.objc.info/about/`
- *Object-Oriented Programming and the Objective-C Language*: `http://toodarkpark.org/computers/objc/`

The rest of this section discusses the basics of compiling Objective-C code with GCC's C compiler, the GCC options that are specific to compiling Objective-C programs, and the highlights of GCC's Objective-C runtime library.

Compiling Objective-C Applications

Objective-C source code files traditionally have the `.m` extension. As an example, Listing 1-4 shows the file hello.m, the traditional hello, world program written in Objective-C.

Listing 1-4. *Hello, World Written in Objective-C*

```
#include <objc/Object.h>
#include <stdlib.h>
#include <stdio.h>

@interface Hello:Object
{
  // Empty because no instance variables are used
}
- (void)msg;
@end

@implementation Hello
- (void)msg
{
    printf("Hello, world!\n");
}
@end

int main(void)
{
    id myHello;  // id is a generic representation of any Objective-C object
    myHello=[Hello new];  // or myHello = [[Hello alloc] init]; for purists

    [myHello msg];

    [myHello free];
    return EXIT_SUCCESS;
}
```

■ **Note** This hello, world example was largely cloned from the example in the *Beginners Guide to Objective-C Programming* by Dennis Leeuw and Pascal Bourguignon, available at `http://gnustep.made-it.com/BG-objc`. This is a friendly and useful introduction to Objective-C that is humorously sprinkled with lines from "Look What They've Done to My Song, Ma," by Melanie Safka. No need for YAHWPIO (yet another hello, world program in Objective-C), though I changed things a bit to make it more readable for this context.

As you can see from this example, the extensions to C provided by Objective-C are defined using keywords that begin with the @ symbol, and the objects and control constructs defined by those extensions are invoked within square brackets in your code. The `@keyword` declarations are typically put into Objective-C header files, but this is just an example, so I've put everything into one file.

Once again, this is not a tutorial on Objective-C programming, but consistent examples are always a good thing. Table 1-2 summarizes the `@keyword` statements that are supported in GCC's Objective-C implementation.

Table 1-2. *@keyword Statements for GCC's Objective-C Support*

Keyword	Definition
`@compatibility_alias`	Enables you to define a class name as equivalent to another class name. For example, `@compatibility_alias foo bar;` tells the compiler that each time it encounters `foo` as a class name, it should replace it with `bar`. The alias must not be the name of an existing class, and the class that you are aliasing must actually exist. This keyword is unique to GCC's Objective-C support.
`@implementation`	Defines the unique methods for a defined class—those that are not simply inherited from its parent. The definition for a class implementation is terminated by an `@end` statement.
`@interface`	Defines a class, its parent (if one exists), and any methods unique to that class. The definition for each class is terminated by an `@end` statement.
`@private`	Used within an `@interface` definition to identify variables that are local to a specific class.
`@protected`	Used within an `@interface` definition to explicitly identify variables that are inherited by all subclasses (which is the default behavior of Objective-C class variables).
`@protocol`	Defines a set of methods that classes can conform to. An `@protocol` definition is terminated by an `@end` statement.
`@public`	Used within an `@interface` definition to identify variables that are visible everywhere.
`@selector`	Identifies a message in Objective-C.
`@synchronized`	Identifies protected areas of code that must be locked during execution.
`@try/@catch/@finally@throw`	Associated with the structured error handling capabilities provided by the NeXT runtime on Mac OS X 10.3 and later systems. See the section later in this chapter titled "Structured Error Handling" for more information.

GCC Options for Compiling Objective-C Applications

This section describes the command-line options that are only meaningful for Objective-C and Objective-C++ programs. Because Objective-C is a superset of C, you can also use C language options when compiling Objective-C programs, as well as the language-independent GCC options that are discussed in Appendix A. Similarly, you can use C++ language options when compiling Objective-C++ applications.

A sample example of compiling an Objective-C program using gcc is the following:

```
gcc -lobjc -o hello hello.m
```

This example compiles the program shown in Listing 1-4, producing an executable named `hello`. If you neglect to include the Objective-C runtime library, you will see error output about undefined references, as in the following:

Listing 1-4. *Sample Error Messages from a Missing Objective-C Library*

```
$ gcc hello.m -o hello
```

```
/tmp/ccxAki59.o: In function `main':
hello3.m:(.text+0x2b): undefined reference to `objc_get_class'
hello3.m:(.text+0x3b): undefined reference to `objc_msg_lookup'
hello3.m:(.text+0x5d): undefined reference to `objc_msg_lookup'
hello3.m:(.text+0x81): undefined reference to `objc_msg_lookup'
/tmp/ccxAki59.o: In function `__objc_gnu_init':
hello3.m:(.text+0xab): undefined reference to `__objc_exec_class'
/tmp/ccxAki59.o:(.data+0x208): undefined reference to `__objc_class_name_Object'
collect2: ld returned 1 exit status
```

You must always specify the `-lobjc` option when linking an Objective-C program using gcc so that gcc links in the Objective-C runtime library. The gcc compiler also provides a number of options that are unique to compiling Objective-C and Objective-C++ programs. These options are shown in Table 1-3.

Table 1-3. *GCC Options for Compiling Objective-C and Objective-C++ Code*

Option	Definition
`-fconstant-string-class=`*`class-name`*	Specifies the name of the class (*class-name*) to use as the name of the constant string class. See the "Constant String Objects" section later in this chapter for more information about defining constant strings and specifying a different name for the constant string class. If the `-fconstant-cfstrings` option is also specified, it will override any `-fconstant-string-class` setting and cause `@"string"` literals to be laid out as constant `CoreFoundation` strings.
`-fgnu-runtime`	Causes gcc to generate object code compatible with the standard GNU Objective-C runtime. This is the default on most systems except for gcc running on the Darwin and Mac OS X platforms.
`-fnext-runtime`	Generates output compatible with the NeXT runtime. This is the default for Darwin and Mac OS X systems. The macro `__NEXT_RUNTIME__` is predefined if this option is used so that applications can identify and target the NeXT runtime.
`-fno-nil-receivers`	Causes gcc to assume that the receiver is valid in all Objective-C message instructions (`[receiver` *`message:arg`*`]`), enabling the use of more efficient entry points in the runtime. This option is only available if you are using the NeXT runtime on Mac OS X 10.3 and later systems.
`-fobjc-exceptions`	Enables syntactic support for structured exception handling in Objective-C, much like that provided by C++ and Java. This option is only available if you are using the NeXT runtime on Mac OS X 10.3 and later systems. See the section titled "Structured Error Handling" later in this chapter for more information.

Table 1-3. *GCC Options for Compiling Objective-C and Objective-C++ Code*

Option	Definition
`-freplace-objc-classes`	Causes gcc to embed a special marker that instructs the loader not to statically link the current object file into the main executable, which enables the object file to be dynamically loaded at runtime using the Mac OS X dynamic loader, dyld. This option is used with the NeXT runtime's Fix-and-Continue debugging mode, where an object file can be recompiled and dynamically reloaded while a program is running, without needing to restart an application. This functionality is only available on Mac OS X 10.3 and later systems.
`-fzero-link`	Suppresses the default behavior of the GNU Objective-C runtime to use calls to `objc_getClass()` to identify class entry points at runtime. When theNeXT runtime is being used, and class names are known at compile time, gcc replaces calls to `objc_getClass()` with static references that are initialized at load time in order to improve runtime performance. This can be useful in Zero-Link debugging mode, since it enables individual class implementations to be modified during program execution.
`-gen-decls`	Creates a file named sourcename.decl that contains interface declarations for all classes encountered in the sourcename.m file and any included files.
`-Wno-protocol`	Issues a warning for every method in a protocol that is not implemented by the class that was declared to implement that protocol and any of its superclasses. The default behavior is to issue a warning for every method not explicitly implemented in that specific class.
`-Wselector`	Causes gcc to display warnings if multiple methods of different types for the same selector are found during compilation. The check is performed on the list of methods in the final stage of compilation, including methods for selectors declared with an `@selector()` expression. These messages are not displayed if compilation terminates due to errors, or if the generic GCC `-fsyntax-only` option was specified and the program is therefore not actually being compiled.
`-Wundeclared-selector`	Causes gcc to display warnings if an `@selector()` expression referring to an undeclared selector is encountered. A selector is considered undeclared if no method with that name has been declared before the `@selector()` expression, either explicitly in an `@interface` or `@protocol` declaration, or implicitly in an `@implementation` section. This option always performs its checks as soon as an `@selector()` statement is encountered, which enforces the Objective-C coding convention that methods and selectors must be declared before being used.
`-print-objc-runtime-info`	Causes gcc to generate a C header to `stdout` that describes the largest structure that is passed by value in an application, if any.

Exploring the GCC Objective-C Runtime

In addition to such standard Objective-C features as dynamic runtime typing, the Objective-C runtime provides support for specific Objective-C constructs that are present in your code or activate during its compilation. This section highlights those features and explains how and why they are used.

Constant String Objects

When compiling Objective-C programs, gcc can generate constant string objects that are instances of the Objective-C runtime's `NXConstantString` class, which is defined in the header file objc/NXConstStr.h. You must therefore include this header file when using Objective-C's constant string objects feature. Constant string objects are declared by defining an identifier consisting of a standard C constant string that is prefixed with the character @, as in the following example:

```
id myString = @"this is a constant string object";
```

The gcc compiler also enables you to define your own constant string class by using the `-fconstant-string-class=`*`class-name`* command-line option. The class that you specify as a new constant string class must conform to the same structure as `NXConstantString`, namely

```
@interface MyConstantStringClass
    {
        Class isa;
        char *c_string;
        unsigned int len;
    }
@end
```

■ **Note** The default class name is `NXConstantString` if you are using GCC's default GNU Objective-C runtime library, and is `NSConstantString` if you are using the NeXT runtime. The discussion in this section focuses on the standard GNU Objective-C runtime.

When creating a statically allocated constant string object, the compiler copies the specified string into the structure's `c_string` field, calculates the length of the string, inserts that value into the length field, and temporarily assigns the value `NULL` to the class pointer. The correct value of that pointer is determined and filled in at execution time by the Objective-C runtime, either with the runtime default or any value that you specified as the value of the `-fconstant-string-class` option.

■ **Note** Because you can incrementally compile Objective-C files and subsequently link the resulting object code, it is possible to specify different constant string classes in different object files using different values of the `-fconstant-string-class` option. While not illegal, this is also not suggested, since at a minimum, this is confusing to everyone but the author of the code and it complicates debugging.

By default, the Objective-C runtime's `NXConstantString` class inherits from the `Object` class. When defining your own constant string class, you can choose to inherit your customized constant string class from a class other than `Object`. Your constant string class doesn't have to provide any

specific methods, but the layout of the data elements in your string class must be compatible with the layout of the standard class definition.

Executing Code Before Main

The GNU Objective-C runtime enables you to execute code before your program enters the main function using the `+load` class load mechanism, which is executed on a per-class and per-category basis. This can be useful to initialize global variables, correctly bind I/O streams, or perform other setup/initialization actions before actually sending a message to a class. The standard `+initialize` mechanism is only invoked when the first message is sent to a class, which could depend on the existing state you would like to set up with the `+load` mechanism.

Though executed on a per-class and per-category basis, the `+load` directive is not overridden by category invocations of that directive—instead, these augment the class's `+load` directives. If a class and a category of that class both invoke the `+load` directive, both methods are invoked. This enables you to do generic initialization on a class and then refine that initialization for specific categories of that class.

Garbage Collection

As discussed in Chapter 11, GCC 4.x enables you to optionally configure and build GCC's Objective-C compiler to use a new memory management policy and associated garbage collector, known to its friends as the Boehm-Demers-Weiser conservative garbage collector. When this garbage collector is used, objects are allocated using a special typed memory allocation mechanism that requires precise information on where pointers are located inside objects. This information is calculated once per class, immediately after the class has been initialized.

In GCC 4.x, the `class_ivar_set_gcinvisible()` runtime function enables you to declare weak pointer references that are essentially hidden from this garbage collector. This function enables you to programmatically track allocated objects, yet still allow them to be collected. Weak pointer references cannot be global pointers, but can only be members of objects. Every type that is a pointer type can be declared a weak pointer, including id, Class, and SEL.

Weak pointers are supported through a new type character specifier represented by the ! character. The `class_ivar_set_gcinvisible()` function adds or removes this specifier to the string type description of the instance variable named as its argument, as in the following example for the interface `foo`:

```
class_ivar_set_gcinvisible (self, "foo", YES);
```

Structured Error Handling

The NeXT Objective-C runtime introduced on Mac OS X 10.3 provides structured error-handling capabilities that should be familiar to most programmers working in object-oriented languages. The rough structure of this error-handling mechanism is the following:

```
@try {

    @throw expr;
    ...
}
@catch (AnObjCClass *exc) {
     ...
    @throw expr;
     ...
```

```
    @throw;
     ...
}
@catch (AnotherClass *exc) {
     ...
}
@catch (id allOthers) {
     ...
}
@finally {
     ...
    @throw expr;
     ...
 }
```

Only pointers to Objective-C objects can be thrown and caught using this scheme. An `@throw` statement may appear anywhere in an Objective-C or Objective-C++ program. When used inside of an `@catch` block, the `@throw` statement need not have an argument, in which case the object caught by the enclosing `@catch` block is rethrown. When an object is thrown, it is caught by the nearest `@catch` clause capable of handling objects of that type, just as in C++ and Java. You can also specify an `@catch(id ...)` clause to catch any exceptions that are not caught by specific `@catch` clauses. If provided, an `@finally` block is executed as soon as the application exists from a preceding `@try ... @catch` section. The `@finally` block is executed regardless of whether any exceptions are thrown, caught, or rethrown inside the `@try ... @catch` section, in the same way that the `finally` clause works in Java.

■ **Note** The Objective-C exception model does not interoperate with C++ exceptions in GCC 4.1 and earlier. You cannot `@throw` an exception from Objective-C and catch it in C++, or catch an exception thrown in C++ in Objective-C.

Synchronization and Thread-Safe Execution

The `-fobjc-exceptions` switch also enables the use of synchronization blocks for thread-safe execution, as in the following example:

```
@synchronized (ObjCClass *guard) {
    ...
}
```

When entering an `@synchronized` block, a thread of execution first checks whether another thread has already placed a lock on the corresponding guard object. If so, the current thread waits until the lock is released before proceeding. Once the guard object is unlocked, the current thread places its own lock on it, executes the code contained in the `@synchronized` block, and then releases the lock. Throwing exceptions out of `@synchronized` blocks will cause the guarding object to be unlocked properly.

■ **Note** Unlike Java, entire methods cannot be marked as `@synchronized` in GCC's Objective-C.

Type Encoding

The GCC Objective-C compiler generates type encodings for all the types encountered in an application. These encodings are used at runtime to find out information about selectors and methods as well as about objects and classes. Table 1-4 shows the type encodings used by GCC's Objective-C and Objective-C++ compilers for basic datatypes, type identifiers, and nonatomic datatypes. Type identifiers are encoded immediately before the types themselves, but are only encoded when they appear in method arguments.

Table 1-4. *Type Encodings Used by GCC for Types and Type Identifiers*

Type	Encoding
array	[followed by the number of elements in the array, followed by the type of those elements; terminated by]
bit-fields	`b`: followed by the starting position of the bit-field, the type of the bit-field, and the size of the bit-field. This differs from the encoding used by traditional Objective-C runtimes, such as the NeXT runtime, in order to allow bit-fields to be properly handled by runtime functions that compute sizes and alignments of types that contain bit-fields.
bycopy	`O`
char	`c`
char *	`*`
class	`#`
const	`r`
double	`d`
float	`f`
id	`@`
in	`n`
inout	`N`
int	`I`
long	`l`
long long	`q`
oneway	`V`
out	`o`
pointer	`^`: followed by the pointed type
SEL	`:`
short	`s`
structure	{ followed by the name of the structure, the = sign, the type of the members, and terminated by }
union	(followed by the name of the structure, the = sign, the type of the members, and terminated by)

Table 1-4. *Type Encodings Used by GCC for Types and Type Identifiers (Continued)*

Type	Encoding
unknown type	?
unsigned char	C
unsigned int	I
unsigned long	L
unsigned long long	Q
unsigned short	S
void	v

■ **Note** Unnamed data structures are encoded using a question mark as the structure name.

CHAPTER 2

Using GCC's C++ Compiler

This chapter discusses typical usage of the GNU Compiler Collection's C++ compiler, g++, focusing on the command-line options and constructs that are specific to g++. GCC's C++ compiler is traditionally installed so you can execute it by either the g++ or c++ commands, just as many installations install cc as a synonym for gcc. This chapter uses g++ in examples and running text because this is the more traditional name of the executable for the GCC C++ compiler.

GCC Option Refresher

Appendix A discusses the options that are common to all of the GCC compilers and the ways to customize various portions of the compilation process. But so you don't have to jump back and forth in the book, this section provides a quick refresher of basic compiler usage as it applies to the GCC C compiler. For detailed information, see Appendix A. If you are new to g++ and just want to get started quickly, you're in the right place.

The g++ compiler accepts both single-letter options, such as `-o`, and multiletter options, such as `-ansi`. Because it accepts both types of options you cannot group multiple single-letter options together, as you may be used to doing in many GNU and Unix/Linux programs. For example, the multiletter option `-pg` is not the same as the two single-letter options `-p -g`. The `-pg` option creates extra code in the final binary that outputs profile information for the GNU code profiler, gprof. On the other hand, the `-p -g` options generate extra code in the resulting binary that produces profiling information for use by the prof code profiler (`-p`) and causes gcc to generate debugging information using the operating system's normal format (`-g`).

Despite its sensitivity to the grouping of multiple single-letter options, you are generally free to mix the order of options and compiler arguments on the gcc command line. That is, invoking g++ as

```
g++ -pg -fno-strength-reduce -g myprog.c -o myprog
```

has the same result as

```
g++ myprog.c -o myprog -g -fno-strength-reduce -pg
```

I wrote that you are generally free to mix the order of options and compiler arguments because, in most cases, the order of options and their arguments does not matter. In some situations, order does matter, if you use several options of the same kind. For example, the `-I` option specifies the directory or directories to search for include files. So, if you specify `-I` several times, gcc searches the listed directories in the order specified.

Compiling a single source file, myprog.cc, using g++ is easy—just invoke g++, passing the name of the source file as the argument.

```
$ g++ myprog.cc
$ ls -l
```

```
-rwxr-xr-x      1 wvh    users         13644 Oct  5 16:17 a.out
-rw-r--r--      1 wvh    users           220 Oct  5 16:17 myprog.cc
```

By default, the result on Linux and Unix systems is an executable file named a.out in the current directory, which you execute by typing **./a.out**. On Cygwin systems, you will wind up with a file named a.exe that you can execute by typing either **./a** or **./a.exe**.

To define the name of the output file that g++ produces, use the `-o` option, as illustrated in the following example:

```
$ g++ myprog.cc -o runme
$ ls -l
```

```
-rw-r--r--      1 wvh    users           220 Oct  5 16:17 myprog.cc
-rwxr-xr-x      1 wvh    users         13644 Oct  5 16:28 runme
```

If you are compiling multiple source files using g++, you can simply specify them all on the gcc command line, as in the following example, which leaves the compiled and linked executable in the file named showdate:

```
$ g++ showdate.cc helper.cc –o showdate
```

If you want to compile these files incrementally and eventually link them into a binary, you can use the `–c` option to halt compilation after producing an object file, as in the following example:

```
$ g++ -c showdate.cc
$ g++ -c helper.cc
$ g++ showdate.o helper.o –o showdate
$ ls -l
```

```
total 124
-rw-r--r--    1 wvh    users         210  Oct 5 12:42 helper.cc
-rw-r--r--    1 wvh    users          45 Oct  5 12:29 helper.h
-rw-r--r--    1 wvh    users        1104 Oct  5 13:50 helper.o
-rwxr-xr-x    1 wvh    users       13891 Oct  5 13:51 showdate
-rw-r--r--    1 wvh    users         208 Oct  5 12:44 showdate.cc
-rw-r--r--    1 wvh    users        1008 Oct  5 13:50 showdate.o
```

■ **Note** All of the GCC compilers "do the right thing" based on the extensions of the files provided on any GCC command line. Mapping file extensions to actions (for example, understanding that files with `.o` extensions only need to be linked) is done via the GCC specs file. Prior to GCC version 4, the specs file was a stand-alone text file that could be modified using a text editor; with GCC 4 and later, the specs file is built-in and must be extracted before it can be modified. For more information about working with the specs file, see "Customizing GCC Using Spec Strings" in Appendix A.

It should be easy to see that a project consisting of more than a few source code files would quickly become exceedingly tedious to compile from the command line, especially after you start adding search directories, optimizations, and other g++ options. The solution to this command-line tedium is the make utility, which is not discussed in this book due to space constraints (although it is touched upon in Chapter 8).

Filename Extensions for C++ Source Files

As mentioned in the previous section, all GCC compilers evaluate filename suffixes to identify the type of compilation that they will perform. Table 2-1 lists the filename suffixes that are relevant to g++ and the type of compilation that g++ performs for each.

Table 2-1. *GCC Filename Suffixes for C++*

Suffix	Operation
`.C`	C++ source code to preprocess.
`.cc`	C++ source code to preprocess. This is the standard extension for C++ source files.
`.cpp`	C++ source code to preprocess.
`.cxx`	C++ source code to preprocess
`.ii`	C++ source code not to preprocess.

A filename with no recognized suffix is considered an object file to be linked. GCC's failure to recognize a particular filename suffix does not mean you are limited to using the suffixes listed previously to identify source or object files. As discussed in Appendix A, you can use the `-x` *`lang`* option to identify the language used in one or more input files if you want to use a nonstandard extension. The `lang` argument tells g++ the input language to use; and for C++, input files can be either c++ (a standard C++ source file) or c++-cpp-output (a C++ source file that has already been preprocessed and therefore need not be preprocessed again).

■ **Note** When any GCC compiler encounters a file with one of the suffixes shown in Table 2-1, it treats the file as a C++ file. Nonetheless, other GCC compilers (such as gcc) do not understand the complete chain of dependencies, such as class libraries, that C++ programs often require, and do not directly know how to compile C++ code. You should therefore always use g++ (or c++ if you are in environments that require this name) to invoke GCC's C++ compiler directly.

Command-Line Options for GCC's C++ Compiler

As explained in Appendix A, many command-line options are common to all of the compilers in the GCC suite. Table 2-2 shows the command-line options that are specific to the g++ compiler.

Table 2-2. *Standard C++ Command-Line Options*

Option	Description
`-fabi-version=n`	Specifies the version of the C++ application binary interface that the compiled code should conform to. For GCC versions 3.4 and greater, the default version is version 2. See the section of this chapter titled "ABI Differences in g++ Versions" for additional information.
`-fcheck-new`	Ensures that the pointer returned by the new operator is not `NULL` before accessing the allocated storage.
`-fconserve-space`	Puts global variables initialized at runtime and uninitialized global variables in the common segment, conserving space in the executable.
`-fdollars-in-identifiers`	Permits the $ symbol in identifiers (the default).
`-fms-extensions`	Allows g++ to omit warnings about nonstandard idioms in the Microsoft Foundation Classes (MFC).
`-fno-access-control`	Disables access checking.
`-fno-const-strings`	Forces g++ to assign string constants to the `char *` type, even though the ISO C++ mandates const `char *`.
`-fno-elide-constructors`	Instructs g++ to always call the `copy` constructor rather than using a temporary to initialize another object of the same type.
`-fno-enforce-eh-specs`	Disables runtime checks for violations of exception handling.
`-ffor-scope`	Limits the scope of variables declared in a `for` initialization statement to the end of the `for` loop. You can specify the `-fno-for-scope` option to force the scope of variables declared in a `for` initialization statement to be the enclosing scope, which is contrary to the ISO standard but was the behavior in older versions of g++ and in many traditional C++ compilers.
`-fms-extensions`	Disables pedantic warnings about constructs from the Microsoft Foundation Classes.
`-fno-gnu-keywords`	Disables the use of `typeof` as a keyword so it can be used as an identifier. You can still use the `__typeof__` keyword. This option is implied when using the `-ansi` option.
`-fno-implement-inlines`	Saves space by not creating out-of-line copies of inline functions controlled by #pragma statements. Using this option will generate linker errors if the `such` functions are not inlined everywhere they are called.
`-fno-implicit-inline-templates`	Saves space by not creating implicit instantiations of inline templates (see `-fno-implicit-templates`).
`-fno-implicit-templates`	Saves space by only creating code for explicit instantiations of out-of-line (noninline) templates.
`-fno-nonansi-builtins`	Disables use of built-ins not required by the ANSI/ISO standard, including `ffs`, `alloca`, `_exit`, `index`, `bzero`, `conjf`, and related functions.
`-fno-operator-names`	Disables the use of the `and`, `bitand`, `bitor`, `compl`, `not`, `or`, and `xor` keywords as synonyms for those operators.

Table 2-2. *Standard C++ Command-Line Options*

Option	Description
`-fno-optional-diags`	Disables nonstandard internal diagnostics, such as when a specific name is used in different ways within a class.
`-fno-rtti`	Disables generation of runtime type identification information for classes with virtual functions.
`-fno-threadsafe-statics`	Causes g++ not to generate the additional code required for the thread-safe initialization of local statics as specified in the C++ ABI. Using this option can reduce code size in applications that do not need to be thread-safe.
`-fno-weak`	Causes g++ not to use weak symbol support, even if it is provided by the linker. This option is intended for use when testing g++, and should not be used in any other situation.
`-fpermissive`	Converts errors about nonconformant code to warnings, allowing compilation to continue.
`-frepo`	Allows template instantiation to occur automatically at link time. Using this option implies the `-fno-implicit-templates` option.
`-fstats`	Displays statistics about front end processing when compilation completes. This option is generally only used by the g++ development team.
`-ftemplate-depth-`*n*	Prevents template instantiation recursion going deeper than the integral value of *n*.
`-fuse-cxa-atexit`	Registers destructors for static objects with static storage duration with the `__cxa_atexit` function rather than `atexit`.
`-fvisibility=`*value*	(GCC 4.02 and later) Causes g++ to not export ELF symbols (Executable and Linking Format, the default binary format used on systems such as Linux and Solaris) from the current object module or library when specifying `hidden` as the value of this option. This substantially reduces the size of resulting binaries and results in performance improvements due to symbol table lookup improvements. However, using this option can also cause problems throwing exceptions across modules with different visibility levels. See the section "Visibility Attributes and Pragmas for GCC C++ Libraries" later in this chapter for more information. If this option is not used, the default visibility value is `default`, which exports all ELF symbols across object files and from libraries.
`-nostdinc++`	Disables searching for header files in standard directories specific to C++.

The g++ compiler recognizes other options specific to C++, but these options deal with optimizations, warnings, and code generation, so we will discuss them in detail in other chapters. In particular, the section of Appendix A titled "Enabling and Disabling Warning Messages" provides general information about using warning options and includes C++-specific warning options. Chapter 5 covers C++-specific optimization options. Table 2-3 summarizes warning options that are specific to C++.

Table 2-3. *Warning-Related Options for C++*

Option	Description
`-Wabi`	Generates a warning when the compiler generates code that is not compatible with the standard C++ ABI. For GCC versions 3.4 and greater, the default version is version 2. See the section of this chapter titled "ABI Differences in g++ Versions" for additional information.
`-Wctor-dtor-privacy`	Generates a warning when all constructors and destructors in a class are private and therefore inaccessible.
`-Weffc++`	Generates a warning for any violation of the stylistic C++ guidelines given in *Effective C++*, Scott Meyers (Addison-Wesley, 2005. ISBN: 0-321-33487-6).
`-Wno-deprecated`	Does not generate warnings for deprecated C++ features or usage.
`-Wno-non-template-friend`	Does not generate warnings when nontemplated friend functions are declared within a template. In the C++ language template specification, a friend must declare or define a nontemplate function if the name of the friend is an unqualified identifier.
`-Wno-pmf-conversions`	Does not generate a warning when a pointer to a member function is internally converted to a standard pointer.
`-Wnon-virtual-dtor`	Generates a warning when a class requires a virtual destructor but declares a nonvirtual destructor. Implied by GCC's `-Wall` option (discussed in Appendix A).
`-Wold-style-cast`	Generates a warning if a traditional C-language-style cast to a nonvoid type is used within a C++ source file.
`-Woverloaded-virtual`	Generates a warning when a function declaration hides virtual functions from a base class.
`-Wreorder`	Generates a warning when the order in which members are initialized in code does not match the order of their use. The g++ compiler automatically records the initialization sequence to reflect the correct initialization order. Implied by `-Wall`.
`-Wsign-promo`	Generates a warning when an overloaded operation chooses to promote a value to a signed type over a conversion to an unsigned type. Versions of g++ prior to version 3.4 would attempt to preserve unsigned types, which is contrary to the C++ standard.
`-Wstrict-null-sentinel`	Generates a warning if an uncast `NULL` is used as a sentinel. A sentinel value is a value that is not a legitimate value for a particular input and is used to indicate a "stopping" value. This is a potential problem because an uncast `NULL` may have different sizes in different compiler implementations.

ABI Differences in g++ Versions

The C++ application binary interface (ABI) is the binary flip side of the application programming interface (API) defined by the C++ datatypes, classes, and methods in the include files and libraries that are provided by a C++ library implementation. A consistent binary interface is required in order for compiled C++ applications to conform to the binary conventions used in associated C++ libraries

and related object files for things such as physical organization, parameter passing conventions, and naming conventions. This consistency is especially critical for specific language support routines, most notably those used for throwing or catching exceptions.

Most recent C++ compilers conform to a specific ABI, which effectively determines the execution environments in which code compiled with that compiler can execute correctly (modulo coding errors, of course—perhaps "execute as written" would be a more precise statement). Beginning with version 3 of the GNU C++ compiler, g++ conforms to an industry-standard C++ ABI as defined in the ABI specification at `http://www.codesourcery.com/cxx-abi/abi.html`. Though this specification was originally written for 64-bit Itanium systems, it provides generic specifications that apply to any platform and is the C++ ABI implemented on platforms such as GNU/Linux and BSD systems.

As with other types of libraries, an existing ABI can be extended through versioning. Versioning enables subsequent library releases to add new symbols and functionality while retaining backward compatibility with previous releases. Obviously, the reverse is not true—binaries linked with the latest release of a library cannot execute against libraries that do not support all of the symbols and functionality provided in the new library.

Versions of g++ prior to version 3.4 use ABI version 1. Versions 3.4 through 4.1 of g++ use ABI version 2. You cannot use libraries compiled with one version of the C++ ABI with an application compiled with another. If you are unsure of the version of the C++ ABI that your version of g++ provides, you can determine this by checking the g++ version using the `g++ --version` command or by extracting the ABI identifier from a pseudo-compile using a command like the following:

```
g++ -E -dM - < /dev/null | awk '/GXX_ABI/ {print $3}'
```

This command will display 102 for g++ compilers using version 1 of the C++ ABI, and 1002 for g++ compilers using version 2 of the C++ ABI.

If you are using version 3.4 or later of g++, you should recompile any of your personal or enterprise code libraries that were compiled with a previous version of g++. If you absolutely must link with or load libraries compiled by a previous version of g++, you can compile your applications using the g++ option `-fabi-version=`*n*, where *n* is the version of the C++ ABI that you want your application to conform to. Using this option to force an older ABI version should really only be viewed as a stopgap—upgrading all the code and libraries that you depend on to the latest version of the C++ ABI is definitely "the right thing."

GNU C++ Implementation Details and Extensions

This book is not a tutorial or reference on writing C++ applications—there are already plenty of books that do that job and do it well. However, when writing C++ code that you will compile using the GCC C++ compiler, you can take advantage of a number of extensions to both C++ through the compiler itself and the Standard C++ library used by g++, libstdc++. This section highlights the most significant extensions that are provided by both, as of the time this book was written, and discusses some differences in behavior you may see between C++ as defined by the C++ specification and the compilation and behavior of C++ code compiled using the GCC C++ compiler.

Attribute Definitions Specific to g++

In addition to the visibility attributes, discussed later in this chapter in the section titled "Visibility Attributes and Pragmas for GCC C++ Libraries," g++ supports two additional attributes that are explicitly designed for use in C++ applications. These are the `init_priority(`*priority*`)` and `java_interface` attributes.

■ **Note** The function attributes discussed in the section of Chapter 1 titled "Declaring Function Attributes" can also be used with C++ applications. When using format attributes such as `nonnull` with C++ methods, you may want to test all parameters (by not specifying one or more argument indices) rather than trying to identify specific parameters, because g++ may internally add parameters to a call because such attributes apply to an entire type.

The init_priority Attribute

The `init_priority(`*priority*`)` attribute enables user control over the order in which objects defined in a namespace scope are initialized. Normally, objects are initialized in the order in which they are defined within a translation unit. The `init_priority` attribute takes a single argument, which is an integer value between 101 and 65,535 (inclusive), where lower numbers indicate a higher priority. For example, in the following pseudo-code, class `MyClass` would be initialized before class `YourClass`:

```
class MyClass
{
  …
};
class YourClass
{
  __attribute__ ((visibility("default"))) void MyMethod();
  …
};
```

To reverse the order in which these are initialized, you could modify this code to look something like the following:

```
class MyClass
{
  __attribute__ ((init_priority(65535)));
  …
};
class YourClass
{
  __attribute__ ((init_priority(101)));
  …
};
```

Only the relative ordering of the priority values that you specify matter—the specific numeric values that you use do not.

The java_interface Attribute

The `java_interface` type attribute informs g++ that the class is a Java interface, and can only be used with classes that are declared within an extern "Java" block. Calls to methods declared in this class are called using the GCC Java compiler's interface table mechanism, instead of the regular C++ virtual function table mechanism.

■ **Tip** Remember that Java's runtime requires a fair amount of initialization that must occur before you call Java methods. When combining C++ and Java code, it is simplest to write your main program in Java so that the initialization is guaranteed to occur before any Java method calls.

C++ Template Instantiation in g++

Templates are one of the most interesting and useful features of C++, reducing code size and the amount of duplicate code, encouraging reusability, and simplifying debugging and code maintenance. Templates are also very useful for enforcing type checking at compile time. For example, using templates eliminates the need for passing a void pointer, since you can define a template and then an instance of the template with the expected types.

The GCC g++ compiler extends the standard ISO template definition by adding three capabilities:

- Support for forward declaration of explicit instantiations by using the `extern` keyword.
- The ability to instantiate the support data required by the compiler for a named template class without actually instantiating it by using the `inline` keyword.
- The ability to only instantiate the static data members of a class without instantiating support data or member functions by using the `static` keyword.

In general, GCC's g++ compiler supports both the Borland and Cfront (AT&T) template models. To support the Borland model of template instantiation and use, g++ provides the `-frepo` option to enable template instances for each translation unit that uses them to be generated at compile time by the preprocessor, which stores them in files with the `.rpo` extension. These files are subsequently included in the compilation process and are collapsed into single instances within compilation units by the linker. To support the Cfront model, g++ internally maintains a repository of template instances and integrates them into code that uses them at link time. The Cfront model requires that code that uses templates either explicitly instantiates them or includes a declaration file so the template definitions are available for instantiation. You can explicitly instantiate the templates you use at any point in your code or in a single file that you include. In the latter case, you may want to compile your code without the `-fno-implicit-templates` option, so you only get all of the instances required by your explicit instantiations.

Function Name Identifiers in C++ and C

GCC compilers predefine two identifiers that store the name of the current function. The `__FUNCTION__` identifier stores the function name as it appears in the source code; while `__PRETTY_FUNCTION__` stores the name *pretty* printed in a language-specific fashion. In C programs, the two function names are the same, but in C++ programs they are usually different. To illustrate, consider the following code from a program named FUNCTION_example.cc:

```
#include <iostream>
using namespace std;

class c {
        public:
        void method_a(void)
        {
              cout << "Function " << __FUNCTION__ <<  " in " << __FILE__ << endl;
              cout << "Pretty Function " << __PRETTY_FUNCTION__ <<  " in "
                      << __FILE__ << endl;
        }
};
```

```
int main(void)
{
        c C;
        C.method_a();
        return 0;
}
```

At runtime, the output is:

```
$ ./a.out
Function method_a in FUNCTION_example.cc
Pretty Function void c::method_a() in FUNCTION_example.cc
```

In C++, __FUNCTION__ and __PRETTY_FUNCTION__ are variables. On the other hand, because they are not macros, #ifdef __FUNCTION__ is meaningless inside a function because the preprocessor does not do anything special with the identifier __FUNCTION__ (or __PRETTY_FUNCTION__).

For a discussion of the same concept in GCC's C compiler, see the section "Function Name As Strings" in Chapter 1.

■ **Note** If you are updating existing code to work with GCC 3.2 and later, these newer GCC versions handle __FUNCTION__ and __PRETTY_FUNCTION__ the same way as __func__, which is defined by the C99 standard as a variable. Versions of GCC's C compiler prior to version 3.2 defined __FUNCTION__ and __PRETTY_FUNCTION__ as string literals, meaning that they could be concatenated with other string literals in character string definitions.

Minimum and Maximum Value Operators

GCC's g++ compiler adds the <? and >? operators, which, respectively, return the minimum and maximum of two integer arguments. For example, the following statement would assign 10 to the min variable:

```
min = 10 <? 15 ;
```

Similarly, the following statement would assign 15 to the max variable:

```
max = 10 >? 15;
```

■ **Tip** Because these operators are language primitives, they can also be overloaded to operate on any class or enum type.

Using Java Exception Handling in C++ Applications

The Java and C++ exception handling models differ, and though g++ tries to guess when your C++ code uses Java exceptions, it is always safest to explicitly identify such situations to avoid link failures. To tell g++ that a block of code may use Java exceptions, insert the following pragma in the current translation unit before any code that catches or throws exceptions:

```
#pragma GCC java_exceptions
```

You cannot mix Java and C++ exceptions in the same translation unit.

Visibility Attributes and Pragmas for GCC C++ Libraries

A traditional problem when writing C++ libraries is that the number of visible ELF symbols can become huge, even when symbols are not used externally and therefore need not be made public. GCC versions 4.02 and later provide the new `-fvisibility=`*value* option and related internal attributes to enable you to control this behavior in a fashion similar to the mechanism provided by `__declspec(dllexport)` in Microsoft's C++ compilers. The new `-fhidden` option takes two possible values: `default`, to continue to export all symbols from the current object file (which is the default value if this option is not specified), and `hidden`, which causes g++ not to export the symbols from the current object module. These two visibility cases are reflected by two new per-function/class attributes for use in C++ applications: `__attribute__ ((visibility("default")))` and `__attribute__ ((visibility("hidden")))`.

By default, ELF symbols are still exported. To prevent ELF symbols from being exported from a specific object file, use the `-fvisibility=hidden` option when compiling that file. This can lead to increased complexity when creating a C++ library using a single Makefile because it requires that you either set manual compilation options for each object file contained in a shared library, or that you add this option to your global compilation flags for the library and therefore do not export any symbols from any of a shared library's component object files. This can be a pain and is actually the wrong thing if you need to make certain symbols visible externally for debugging purposes or for catching exceptions for throwable entities within a library.

■ **Caution** Being able to catch an exception for any user-defined type in a binary other than the one that threw the exception requires a `typeinfo` lookup. Because `typeinfo` information is part of the information that is hidden by the `-fvisibility=hidden` option or the `visibility __attribute__` attribute specification, you must be very careful not to hide symbols for any class or method that can throw exceptions across object file boundaries.

The best mechanism for making selected symbols visible is through a combination of using the visibility attributes and the `-fvisibility=hidden` command-line option. For any structs, classes, or methods whose symbols must be externally visible, add the `__attribute__ ((visibility("default")))` attribute to their definition, as in the following example:

```
class MyClass
{
  int i;
  __attribute__ ((visibility("default"))) void MyMethod();
  …
};
```

You can then add the `-fvisibility=hidden` option to the generic Makefile compilation flags for all of your object files, which will cause symbols to be hidden for all structs, classes, and methods that are not explicitly marked with the default visibility attribute. When the source module containing this code fragment is compiled, symbols are hidden by default, but those for the `MyMethod()` method would be externally visible.

A slightly more elegant and memorable solution to symbol visibility is to define a local visibility macro in a global header file shared by all of the object modules in a library, and then put that macro in front of the declaration for any class or method definition whose symbols you do not want to export. In the converse of the previous example, if you want to export symbols by default but restrict those for selected classes and methods, this might look something like the following:

```
#define LOCAL    __attribute__ ((visibility("hidden")))

class MyClass
{
  int i;
  LOCAL void MyMethod();
  …
};
```

The source module containing this sample code could be compiled without using the `-fvisibility=`*value* option and would export all symbols from `MyClass` with the exception of those in the method `MyMethod()`.

A pragma for the new visibility mechanism is currently supported, but will probably go away in future versions. New code should use `visibility __attribute__`, but existing class and function declarations can quickly be converted using the visibility pragma, as in the following example:

```
extern void foo(int);
#pragma GCC visibility push(hidden)
extern void bar(int);
#pragma GCC visibility pop
```

In this example, symbols would still be exported from `foo()` but not from `bar()`. You must also be especially careful not to use this pragma with user-defined types that can throw exceptions across object file boundaries due to the fact that `typeinfo` information is part of the symbol information that is hidden, as explained earlier.

CHAPTER 3

Using GCC's Fortran Compiler

It used to be that your first computer science class taught you how to use some variant of FORTRAN, the FORmula TRANslation language. It also used to be the case that the machines you learned it on had no GUIs, and they serviced large numbers of users with 256KB of RAM. (Honest!) Times have obviously changed, so why would anyone care about Fortran nowadays?

Note Since *Fortran* is pronounced as a word, rather than saying each letter, I will follow the standard by writing it in initial-cap style rather than in all caps.

If you're reading this, you certainly care or are at least curious about Fortran and GCC. Regardless of your motivation, there are a number of good reasons to use Fortran today. Since you are probably less interested in philosophy than in actually compiling and debugging some Fortran code, we'll just hit the highlights:

- A vast amount of legacy Fortran is already available and is both time tested and programmer approved. Unless you need to make substantial changes to existing code, why bother to change it at all (and risk breaking something) if it already works?
- Fortran is a powerful, low-overhead language that was originally designed for scientific number crunching, and still does a great job at it. It offers a number of language constructs that are both convenient and highly tuned, such as built-in support for exponentiation of various data types, complex arithmetic, and very small floating-point numbers, and the ability to pass variable dimension arrays as arguments to subroutines. It is therefore well-suited to operations such as huge matrix manipulation operations and other calculation-intensive operations.
- Fortran is a simple language. Fortran code is often much easier to debug than code in more modern languages because it typically relies on simple data structures and array indices rather than C constructs such as *foo, &foo, **foo, and so on, which can resemble line noise or TECO commands.

Some more creative explanations for the continued existence of Fortran can be found in the Fortran FAQ, which is available on sites such as `http://www.faqs.org/faqs/fortran-faq/`. However, the fact that this FAQ hasn't been updated since January 1997 is a statement in itself. The next section has some more modern Fortran references.

For a contrarian view, see the petition to retire Fortran at `http://www.fortranstatement.com/cgi-bin/petition.pl`.

Fortran History and GCC Support

Though you're probably reading this chapter because you actually want to use the GNU Fortran compiler, a bit of background on Fortran and GNU Fortran support is always useful.

FORTRAN IS FUN

Just to show that Fortran can indeed be fun, here's a classic Fortran joke:

GOD is REAL (unless declared INTEGER)

This joke had them rolling in the aisles in the 1960s. But if it just leaves you scratching your head, the idea behind this joke is that in Fortran, variables with names that begin with the letters *I* through *N* are considered to be integer variables, while variables beginning with any other letters are considered to be real variables (unless explicitly associated with a specific type using the IMPLICIT statement). I'll bet you're laughing now!

Another fun bit of Fortran humor is available in the discussion of the COME FROM statement, originally published in the December 1973 issue of *Datamation*, but now available online at `http://www.fortran.com/come_from.html`.

Fortran was the first high-level language to escape from research labs and academia into actual commercial use. Fortran was invented in 1954 by a team of researchers at IBM led by John Backus. Originally developed for the IBM 701 and known as Speedcoding, Fortran was made commercially available for the IBM 704, which Backus also helped to design, in 1957. (A PDF version of the original manual for Fortran on the IBM 704 is available at `http://www.fortran.com/FortranForTheIBM704.pdf`.) In recognition of his contributions to computer science by inventing Fortran, Backus was awarded the National Academy of Engineering's Charles Stark Draper Prize in 1993, which is the highest prize awarded in engineering in the United States.

High-level languages are a given today, but in the mid-1950s, the idea of a computer language that had a logical, abstract syntax that was independent of the architecture or implementation of a specific computer or instruction set was still fairly revolutionary. This is not to say that Fortran was invented in 1954 and has continued unchanged to this day. Quite the contrary. The following is a capsule summary of the different "official" versions of Fortran that have been used as standards from 1957 until today:

- *FORTRAN I - (1957, IBM)*: Became the original version of Fortran.
- *FORTRAN II - (1958, IBM)*: Introduced subroutines, functions, and common block. Featured a primordial three-way `if` statement based on whether a numeric expression was negative, zero, or positive.
- *FORTRAN III - (1958, IBM)*: Designed in 1958 but never officially released, largely because its support for inline assembler code was viewed as a violation of the high-level language concept. Introduced Boolean expressions, which actually saw the light of day in FORTRAN IV.
- *FORTRAN IV - (1961, IBM)*: Introduced Boolean expressions and actual `if` tests based on the Boolean value of an expression.
- *FORTRAN 66 - (1966, ANSI Standard X3.9-1966)*: Became the first official Fortran standard—36 pages of Fortran fun.
- *FORTRAN 77 - (1977, ANSI Standard X3.9-1978)*: Provided an improved I/O facility and new features for querying, opening, and closing files. Introduced the if-then-else-end `if` construct and the character data type for text.

- *Fortran 82, Fortran 8x, Fortran 88*: Helped fill the dangerously long gap between FORTRAN 77 and Fortran 90 but were aborted for not producing a new standard.
- *Fortran 90 - (1990, ANSI Standard X3.198-1992, ISO Standard 1539:1991)*: Introduced "free-form" Fortran that was liberated from the punch card–oriented formatting rules of earlier versions of the language. Also introduced dynamic memory allocation, recursion, CASE statements, array operations, abstract data types, operator overloading, and pointers.
- *Fortran 95 - (1995, ISO/IEC Standard 1539-1:1997)*: Introduced the FORALL construct and user-defined pure and elemental functions. Essentially a minor revision of Fortran 90.
- *Fortran 2003 - (2003, ISO/IEC Standard 1539-1:2004)*: Introduced support for IEEE floating-point arithmetic, exception handling, object-oriented programming, and improved interoperability with the C language.

A related standard for anyone developing modern Fortran code is IEEE's POSIX FORTRAN 77 Language Interfaces standard (IEEE Std 1003.9-1992). It defines a standardized interface for accessing the system services of IEEE Std 1003.1-1990 (POSIX.1), and describes routines to access capabilities that are not directly available in FORTRAN 77. See `http://standards.ieee.org/reading/ieee/std_public/description/posix/1003.9-1992_desc.html` for a table of contents. Unfortunately, the IEEE folks charge money for copies of their standards. (They are not alone—so does ISO.) You can view this standard at `http://standards.ieee.org/reading/ieee/std/posix/1003.9-1992.pdf` if you're an IEEE standards online subscriber—otherwise it will redirect you to a page explaining where to send your hard-earned cash.

The first GNU compiler for Fortran was the g77 compiler by James Craig Burley and a host of others. As the name suggests, g77 is a FORTRAN 77–compliant Fortran compiler. For a variety of reasons, work on actively enhancing g77 ceased around the release of GCC 3.0. The decision was made to write a new Fortran 95–compliant compiler, known as gfortran, rather than do a substantial rewrite of g77 to bring it into line with the latest GCC internals. Because FORTRAN 77 is essentially a proper subset of Fortran 95, most existing code that you can compile with g77 can also be compiled with gfortran. Later in this chapter, the section "Classic GNU Fortran: The g77 Compiler" discusses the status of g77 and explores issues in migrating and compiling code that was written for use with g77.

■ **Note** At the time this book was written, GCC's gfortran compiler was largely, but not completely, compliant with Fortran 95. Work toward this goal is actively in progress. Currently, there are still known problems with using the `ENTRY` and the `NAMELIST` declarations, as well as with more advanced uses of `MODULES`, `POINTERS`, and `DERIVED TYPES`. If you are trying to compile Fortran 95 code that you can't get to work with gfortran, file a bug with the GNU folks (`http://gcc.gnu.org/bugs.html`). If the problem you have encountered is a show-stopper for you, see the section later in this chapter titled "Alternatives to gfortran and g77."

Compiling Fortran Applications with gfortran

This section discusses how to compile applications using gfortran and provides sample code that highlights some of the features of gfortran and the changes in Fortran over the last *n* years. It begins by highlighting compilation options that are shared with other compilers in the GCC family.

Common Compilation Options with Other GCC Compilers

Compiling Fortran applications with gfortran is quite straightforward for anyone familiar with other compilers in the GCC family. Because gfortran depends on many of the core components of GCC, gfortran shares most of the standard command-line options for GCC compilers. Listing 3-1 shows

the output from the gfortran `--help` command, which lists the standard GCC options that are available in gfortran. These general options are discussed in Appendix A of this book. Command-line options that are specific to gfortran (and do not appear in Listing 3-1) are discussed in the section "Command-Line Options for gfortran," later in this chapter.

Listing 3-1. *Output from the gfortran --help Command*

```
Usage: gfortran [options] file..
Options:
  -pass-exit-codes         Exit with highest error code from a phase
  --help                       Display this information
  --target-help               Display target specific command line options
  (Use '-v --help' to display command line options of sub-processes)
  -dumpspecs               Display all of the built in spec strings
  -dumpversion             Display the version of the compiler
  -dumpmachine           Display the compiler's target processor
  -print-search-dirs        Display the directories in the compiler's search path
  -print-libgcc-file-name  Display the name of the compiler's companion library
  -print-file-name=<lib>   Display the full path to library <lib>
  -print-prog-name=<prog>  Display the full path to compiler component <prog>
  -print-multi-directory   Display the root directory for versions of libgcc
  -print-multi-lib             Display the mapping between command line options and
                           multiple library search directories
  -print-multi-os-directory Display the relative path to OS libraries
  -Wa,<options>            Pass comma-separated <options> on to the assembler
  -Wp,<options>            Pass comma-separated <options> on to the preprocessor
  -Wl,<options>             Pass comma-separated <options> on to the linker
  -Xassembler <arg>     Pass <arg> on to the assembler
  -Xpreprocessor <arg>     Pass <arg> on to the preprocessor
  -Xlinker <arg>             Pass <arg> on to the linker
  -combine                    Pass multiple source files to compiler at once
  -save-temps               Do not delete intermediate files
  -pipe                          Use pipes rather than intermediate files
  -time                          Time the execution of each subprocess
  -specs=<file>             Override built-in specs with the contents of <file>
  -std=<standard>        Assume that the input sources are for <standard>
  --sysroot=<directory>    Use <directory> as the root directory for headers
                           for headers and libraries
  -B <directory>            Add <directory> to the compiler's search paths
  -b <machine>             Run gcc for target <machine>, if installed
  -V <version>              Run gcc version number <version>, if installed
  -v                               Display the programs invoked by the compiler
  -###                          Like -v but options quoted and commands not executed
  -E                             Preprocess only; do not compile, assemble or link
  -S                            Compile only; do not assemble or link
  -c                             Compile and assemble, but do not link
  -o <file>                    Place the output into <file>
  -x <language>          Specify the language of the following input files
                            Permissible languages include: c c++ assembler none
                           'none' means revert to the default behavior of
                           guessing the language based on the file's extension
 Options starting with -g, -f, -m, -O, -W, or --param are automatically
 passed on to the various sub-processes invoked by gfortran.  In order to pass
 other options on to these processes the -W<letter> options must be used.
```

Sample Code

Listing 3-2 uses a Fortran program for displaying numbers in the Fibonacci sequence. This program prints the first eight values in the Fibonacci sequence. I didn't write this code, but it serves as a useful starting point for the discussion of compiling Fortran code in the next section.

Listing 3-2. *Sample Fortran Code fib.f90*

```
!
! Hacked Fibonacci program from the web:
!    http://cubbi.org/serious/fibonacci/fortran.html
!
PROGRAM MAIN
      DO 200, K=0,7
         I=K
         J=K+1
         CALL F(I)
         PRINT *,J,'th Fibonacci number is',I
200   CONTINUE
      END PROGRAM
!
! Subroutine F(I) calculates the I'th Fibonacci number
! It uses ALGORITHM 2A-3: DATA STRUCTURE - SIMPLE LIST
!
! Modified to begin with 0 - wvh
!
      SUBROUTINE F(I)
      DIMENSION A(I+1)
      A(1)=0; A(2)=1
      DO 1, J=3,I+1
          A(J)=A(J-1)+A(J-2)
1     CONTINUE
      I=A(I+1)
      RETURN
      END SUBROUTINE
```

The code shown in Listing 3-2 is vanilla Fortran code that compiles successfully using most Fortran 90 and Fortran 95 compilers. The name of the file containing this code is `fib.f90`. I've used Fortran 90–compliant syntax in Listing 3-2 in order to highlight some of the features of gfortran, as explained in the next section.

■ **Tip** You can download the source code for this example and other examples throughout this book from the Apress Web site at `http://apress.com/book/download.html`.

Compiling Fortran Code

The most basic gfortran compilation command is the following:

```
$ gfortran fib.f90
```

As with all GCC compilers, this command produces a binary named `a.out` by default. The output of executing this program is the following:

```
$ ./a.out
```

```
           1 th Fibonacci number is           0
           2 th Fibonacci number is           1
           3 th Fibonacci number is           1
           4 th Fibonacci number is           2
           5 th Fibonacci number is           3
           6 th Fibonacci number is           5
           7 th Fibonacci number is           8
           8 th Fibonacci number is          13
```

You may have noted that this isn't the cleanest output. Don't worry, that's by design. I'll clean this up later in this section as part of an exercise to modernize the code in general.

To identify the name of a specific output file, use the standard GCC `-o` filename option, as in the following example:

```
$ gfortran -o fib fib.f90
```

This produces the same executable, but gives it the name `fib` instead of the default name `a.out`.

gfortran uses the extension of your input program to identify the version of Fortran that your code conforms to. The following extensions have special meaning to the gfortran compiler:

- `.f`: Interpreted as fixed-form FORTRAN 77 code, with no preprocessing needed
- `.F`: Interpreted as fixed-form Fortran source, but performs preprocessing
- `.f90`: Interpreted as free-form Fortran 90 code, with no preprocessing needed
- `.F90`: Interpreted as free-form Fortran 90 code, but performs preprocessing
- `.f95`: Handled in the same way as code with the `.f90` extension
- `.F95`: Handled in the same way as files with the `.F90` extension

To demonstrate the implications of your choice of file extension, let's copy `fib.f90` to `fib.f` to see if gfortran behaves differently when it treats the input file as a FORTRAN 77 file:

```
$ cp fib.f90 fib.f
$ gfortran -o fib fib.f
```

```
 In file wvh_fib.f:17
 PROGRAM MAIN
1
Error: Non-numeric character in statement label at (1)
 In file wvh_fib.f:17
 PROGRAM MAIN
1
Error: Unclassifiable statement at (1)
```

Well, that was certainly different! In this case, gfortran complained because the PROGRAM declaration didn't start in the seventh column, as is required in FORTRAN 77. Listing 3-3 shows the same code with this correction made.

Listing 3-3. *FORTRAN 77–Compliant Code Example*

```
!
! Hacked Fibonacci program from the web:
!    http://cubbi.org/serious/fibonacci/fortran.html
!
      PROGRAM MAIN
      DO 200, K=0,31
         I=K
         J=K+1
         CALL F(I)
         PRINT *,K,'th Fibonacci number is',I
200   CONTINUE
      END PROGRAM
!
! Subroutine F(I) calculates the I'th Fibonacci number
! It uses ALGORITHM 2A-3: DATA STRUCTURE - SIMPLE LIST
!
! Modified to begin with 0 - wvh
!
      SUBROUTINE F(I)
      DIMENSION A(I+1)
      A(1)=0; A(2)=1
      DO 1, J=3,I+1
          A(J)=A(J-1)+A(J-2)
1     CONTINUE
      I=A(I+1)
      RETURN
      END SUBROUTINE
```

This program now compiles successfully as a FORTRAN 77 program:

```
$ gfortran -o fib fib.f
```

As an alternative to modifying the code to be FORTRAN 77–compliant, you could also use the `-ffree-form` option, which tells the gfortran compiler to ignore any formatting requirements that are associated with a specific version of Fortran, as in the following example:

```
$ gfortran -o fib -ffree-form fib.f
```

Modernizing the Sample Fortran Code

The sample code shown in Listings 3-2 and 3-3 is interesting but somewhat primitive, and doesn't really match the Fibonacci code we use elsewhere in this book. Modifying the code to behave in the same way as our Fibonacci code written in C requires changing the code so it can interpret command-line arguments, display a usage message, and exit if none are provided. This is somewhat outside the scope of normal Fortran code, but it's easy enough to do, thanks to some of the extensions that are built into gfortran. These extensions are explained in detail in the section "gfortran Intrinsics and Extensions," later in this chapter.

Listing 3-4 shows an updated version of the same program called `fib_cmdline.f90`, with command-line parsing and a usage message added.

Listing 3-4. *Modernized Sample Code with Command-Line Arguments*

```
!
! Simple program to print a certain number of values
!  in the Fibonacci sequence.
!
! Lifted the F(I) routine from:
!
!    http://cubbi.org/serious/fibonacci/fortran.html
!
! Added command-line arg parsing, corrected fib
!  value sequence in F SUBROUTINE to begin with 0, etc.
!
! NOTE: This code uses the intrinsic functions/subroutines
!       iargc() and getarg() for command-line parsing.
!
!  - wvh
!
PROGRAM MAIN
      integer i
      integer iargc
      integer numarg
      character ( len = 80 ) arg
      numarg = iargc ( )
       if (numarg .ne. 0) then
         call getarg ( 1, arg )
         read(arg,*) j
         DO 100, K=0,j-1
            i = k
            call f(i)
            print *,i
100      continue
      else
         print *,'Usage: fib_cmdline num-of-sequence-values-to-print'
      end if
      END PROGRAM
!
! Subroutine F(I) calculates the I'th Fibonacci number
! It uses ALGORITHM 2A-3: DATA STRUCTURE - SIMPLE LIST
!
      SUBROUTINE F(I)
      DIMENSION A(I+1)
      A(1)=0; A(2)=1
      DO 1, J=3,I+1
          A(J)=A(J-1)+A(J-2)
1     CONTINUE
      I=A(I+1)
      RETURN
      END SUBROUTINE
```

Note that this program uses the same `F(I)` subroutine, but that the main section of the program has been rewritten to use the `iargc()` and `getarg()` extensions. The `iargc()` extension is a function that returns the number of command-line arguments provided when the program is executed. The code in Listing 3-4 also uses the `getarg()` extension to return the first command-line argument as a

character scalar, and uses the standard Fortran `write` command to convert this from a sequence of characters to an integer.

Fortran classicists will also notice that every possible liberty has been taken with the code, mixing both uppercase and lowercase variable names and subroutine calls to highlight the case-insensitive nature of gfortran. This program compiles cleanly using the following command:

```
$ gfortran -o fib_cmdline fib_cmdline.f90
```

Running this command with no arguments provides the following output:

```
$ ./fib_cmdline
```

```
 Usage: fib_cmdline num-of-sequence-values-to-print
```

Running it with the command-line argument 8 displays the first eight values in the Fibonacci sequence:

```
$ ./fib_cmdline 8
```

```
           0
           1
           1
           2
           3
           5
           8
          13
```

Note that the output of the program is still Fortran style, with one output statement per line. This is still pretty darn close to the C version used elsewhere in this book.

The extensions used in this code are available during compilation because code compiled by gfortran allows the use of built-in extensions by default. This is known as "conforming to the GNU Fortran standard," and even has its own command-line option, `-std=gnu`, which is active by default. gfortran supports its own GNU standard, and also supports the Fortran 95 standard, since gfortran is designed as a Fortran 95 compiler. To see what happens when trying to compile this code as standard Fortran 95 code, try using the `-pedantic` and `-std=f95` options, which report violations of the Fortran 95 standard and the use of extensions, as in the following example:

```
$ gfortran -o fib_cmdline fib_cmdline.f -pedantic -std=f95
```

```
/tmp/ccMQpqLo.o(.text+0x18): In function 'MAIN__':
fib_cmdline.f: undefined reference to 'iargc_'
/tmp/ccMQpqLo.o(.text+0x38):fib_cmdline.f: undefined reference to 'getarg_'
collect2: ld returned 1 exit status
```

This output identifies the `iargc()` and `getarg()` functions as not being part of the Fortran 95 standard. However, extensions such as these have been part of Fortran for quite a while, beginning with FORTRAN 77, and are detailed in each Fortran specification. gfortran actually provides a rich set of usable extensions that can significantly simplify Fortran development, especially the mathematical or scientific code for which Fortran is famous. For more information about the extensions to Fortran 95 that are provided by gfortran, see the later section titled "gfortran Intrinsics and Extensions."

Command-Line Options for gfortran

gfortran provides a rich set of command-line options. These options can be generally grouped into five different classes:

- *Code generation*: The type and characteristics of the output of compiling or preprocessing your Fortran programs (code generation) including warning messages
- *Debugging*: Options that can help in debugging your Fortran applications
- *Directory search*: Options that control the directories searched for precompiled subroutines or libraries
- *Fortran dialect*: The characteristics of the Fortran program you are compiling and the Fortran standard to which it conforms
- *Warnings*: The type and severity of warning messages that can be produced when using gfortran

The following sections discuss each of the classes of command-line options in detail.

Code Generation Options

This section describes options that control gfortran code generation. Most of them have both positive and negative forms, such as `-funderscoring` and `-fno-underscoring`.

■ **Note** The options listed in this section are the default settings for gfortran, which means that these options are invoked unless explicitly disabled, even if they are not physically specified on the gfortran command line.

`-fbounds-check`: Causes gfortran to generate runtime checks for array subscripts against the declared minimum and maximum values. Specifying this option also causes gfortran to check the array indices for assumed and deferred shape arrays against the actual allocated bounds.

`-ff2c`: Causes the gfortran compiler to generate code that is designed to be compatible with code generated by g77 and f2c. Using this option implies the `-funderscoring` option. It also implies the `-fsecond-underscore` option, unless `-fno-second-underscore` is explicitly requested. This does not affect the generation of code that interfaces with the libgfortran library.

`-fmax-stack-var-size=`*n*: Specifies the size in bytes of the largest predeclared array that can be put on the stack. The default value for *n* is 32768.

`-fno-underscoring`: Uses the names of entities specified in the Fortran source file, rather than appending underscores to them. Using this option simplifies debugging and interfacing gfortran code to other languages because it allows the direct specification of user-defined names for variables, functions, and subroutines. Even if this option is specified, the entry point for the sample program is still named MAIN__.

■ **Note** You should use the `-ff2c` option if you want object files compiled with gfortran to be compatible with object code created with f2c and g77. Using the `-ff2c` option is currently the default behavior of gfortran, but this may change in the future. For more information about f2c, see the section later in this chapter titled "The f2c Fortran-to-C Conversion Utility." For more information about the g77 compiler, see the section later in this chapter titled "Classic GNU Fortran: The g77 Compiler."

`-fpackderived`: Tells gfortran to pack derived type members as closely as possible. Code compiled with this option is likely to be incompatible with code compiled without this option and may execute slower.

`-frepack-arrays`: Adds code to functions that pass assumed shape arrays as parameters. This code repacks the data into a contiguous block at runtime. This may result in faster access to array data, but can introduce significant overhead to the function call, especially if the passed data is discontiguous and the data must be repacked.

`-fsecond-underscore`: Causes gfortran to append two underscores to internal entity names that do not already end in an underscore, and to append a single underscore to the names of internal entities that already end in an underscore. By default, gfortran appends an underscore to external names (required for compatibility with f2c and g77). This option has no effect if the `-fno-underscoring` option is used, and is implied by the `-ff2c` option.

Debugging Options

The gfortran compiler provides a single debugging option beyond the standard set of default GCC debugging options. See the "GCC Option Reference" section of Appendix A for a list of all GCC debugging options.

`-fdump-parse-tree`: Tells gfortran to dump its internal parse tree before starting code generation. This option is primarily used by gfortran developers when debugging gfortran itself.

Directory Search Options

The options in this section tell the gfortran compiler where to search for external Fortran routines, precompiled Fortran modules, libraries, and so on.

`-Idir`: Tells the gfortran compiler where to look for external Fortran code specified with the INCLUDE directive, precompiled modules specified in USE statements, and so on.

`-Jdir`: An alias for the `-Mdir` option to avoid conflicts with existing GCC options.

`-Mdir`: Tells the gfortran compiler where to put the .mod files produced when separately compiling modules. The specified directory is also added to the list of directories that is searched for modules specified in a USE statement. The default is the current directory.

Fortran Dialect Options

The options in this section tell the compiler what format it should expect to encounter in your input Fortran files and how to interpret lines in the input file and control the handling of various data types and characters encountered in the input file.

`-fdefault-double-8`: Sets the width of the `DOUBLE PRECISION` type to 8 bytes.

`-fdefault-integer-8`: Sets the width of the default integer and logical types to 8 bytes.

`-fdefault-real-8`: Sets the width of the default real type to 8 bytes.

`-fdollar-ok`: Allows the use of the dollar sign, "$", as a valid character in variable or subroutine names.

`-ffixed-form`: Specifies that the input Fortran program code was written using the punch-card oriented layout conventions used in FORTRAN 77 and earlier standards.

`-ffixed-line-length=`*n*: Sets the column after which characters are ignored in each line of the input file, and in which each line in the input file will be padded with spaces. Popular values for *n* are 0, which means that the entire line is meaningful and should not be padded (see the `-fixed-line-length-none` option), 72 (the traditional FORTRAN standard, and the default value), 80 (standard card images that use FORTRAN 77 and earlier layout conventions), and 132 (corresponding to the higher-density cards used in some later keypunch machines).

`-ffixed-line-length-none`: Tells the compiler that every line in an input file should be read as is, with no padding or special assumptions. This is the default.

`-ffree-form`: Specifies that the input Fortran program code was written using the free-form layout introduced in Fortran 90. This means that the input code does not follow the older, punch-card oriented layout conventions used in FORTRAN 77 and earlier standards.

`-fimplicit-none`: Specifies that no implicit typing is allowed, unless overridden by explicit IMPLICIT statements in the Fortran code. This is the equivalent of adding `IMPLICIT NONE` to the start of every subroutine.

`-fmax-identifier-length=`*n*: Specifies the maximum length of an identifier. Typical values are 31 (conforming to the Fortran 95 standard) and 63 (conforming to the Fortran 2003 standard).

`-fno-backslash`: Changes the interpretation of the backslash symbol from the C-style escape character to being treated as the literal single backslash character.

`-std=std`: Specifies that the input Fortran code conforms to a given standard. Possible values for std are gnu and f95. If `-std=f95` is not specified, `-std=gnu` is assumed.

■ **Note** Using the `-std=f95` option in conjunction with the `-pedantic` and `-Wnonstd-intrinsic` options can be useful for checking that your Fortran code is portable, since using these options together will identify GNU intrinsics and extensions that may not be supported in other Fortran compilers.

Warning Options

The options in this section tell the compiler the types of things that it should check for in Fortran source files and how to handle any warning messages triggered by the program code:

`-fsyntax-only`: Checks the code for syntax errors without producing an executable.

`-pedantic`: Causes the compiler to issue warnings for the use of extensions to Fortran 95. This option also applies to any C-language constructs that appear in Fortran source files. If you are using this option to attempt to enforce adherence to the Fortran 95 standard, you should also specify the `-std=f95` option in order to more strictly identify conformance problems.

`-pedantic-errors`: Causes any warnings that would have been identified by the `-pedantic` option to be treated as errors, halting compilation.

`-w`: Suppresses all warning messages.

`-W`: Activates other types of warnings, such as the `-Wuninitialized` option if optimization is specified via `-O`. The specific warnings activated by this option are likely to change with different versions of gfortran.

`-Waliasing`: Issues warnings for possible misuse of subroutine arguments, as when the same variable is accidentally passed to a subroutine as both an input and an output value.

`-Wall`: Enables a set of commonly used warning options for code constructs that the gfortran folks recommend avoiding: `-Wunused-labels`, `-Waliasing`, `-Wsurprising`, `-Wnonstd-intrinsic`, and `-Wline-truncation`.

`-Wconversion`: Issues warnings about implicit conversions between different datatypes.

`-Werror`: Causes any warnings that would have been issued by gfortran to be treated as errors, halting compilation.

`-Wimplicit-interface`: Issues warnings when subroutines are called without a local or explicitly declared external interface.

`-Wnonstd-intrinsic`: Issues warnings if the Fortran code uses an intrinsic that does not belong to the specified standard using the `-std` option.

`-Wsurprising`: Issues warnings when suspicious code constructs are encountered. While technically legal, these usually indicate that an error has been made. This currently catches problems such as when an `INTEGER SELECT` construct has a case that can never be matched because its lower value is greater than its upper value, and when a `LOGICAL SELECT` construct has three or more CASE statements.

`-Wunderflow`: Produces warnings when numerical constant expressions are encountered that have values less than the minimum allowable value of a specified type.

`-Wunused-labels`: Generates warnings for any labels that are present in the Fortran code but which are never referenced.

gfortran Intrinsics and Extensions

Intrinsic functions (intrinsics) and extensions are the terms used to describe additional functions that are built into gfortran. Some intrinsics and extensions are true functions in the C language sense and can therefore be called directly in any code compiled using gfortran without using the standard `CALL` mechanism. They can also return a value like standard C function calls, rather than returning a value in a parameter passed to the subroutine.

Table 3-1 shows the extensions that are provided by gfortran in GCC 4.1, which was the latest version at the time this book was written.

Table 3-1. *Intrinsic Functions and Extensions in gfortran*

Extension	Invoked As	Parameters	Description
ABORTCALL	`Abort()`	NONE	Prints a message and potentially causes a core dump via the standard Linux `abort(3)` function.
ABS	`Abs(A)`	`A`: INTEGER or REAL	Returns the absolute value of `A`.
ACHAR	`AChar(I)`	`I`: CHARACTER*1	Returns the ASCII character corresponding to the code specified by `I`.
ACOS	`ACos(X)`	`X`: REAL	Returns the arc-cosine (inverse cosine) of `X` in radians.

Table 3-1. *Intrinsic Functions and Extensions in gfortran (Continued)*

Extension	Invoked As	Parameters	Description
ADJUSTL	`AdjustL(STR)`	`STR`: CHARACTER scalar	Returns `STR` with leading spaces removed.
ADJUSTR	`AdjustR(STR)`	`STR`: CHARACTER scalar	Returns `STR` with trailing spaces removed.
AIMAG	`Aimag(Z)`	`Z`: COMPLEX	Returns a REAL value containing the imaginary part of `Z`.
AINT	`Aint(X,KIND)`	`X`: REAL; `KIND`: Integer initialization expression (optional)	Returns `X` with the fractional portion of its magnitude truncated and its sign preserved, possibly converted based on the `KIND` expression.
ALL	`All(MASK,DIM)`	`MASK`:LOGICAL; `DIM`: INTEGER (optional)	Returns `TRUE` if all values in `MASK` are `TRUE`. The number of values in `MASK` to check is `DIM` if `DIM` is specified.
ALLOCATED	`Allocated(X)`	`X`: Allocatable array	Returns `TRUE` if `X` is allocated, and FALSE otherwise.
ANINT	`Anint(X[,KIND])`	`X`: REAL; `KIND`: Integer initialization expression (optional)	Returns `X` rounded to the nearest whole number, possibly converted based on the `KIND` expression.
ANY	`Any(MASK[,DIM])`	`MASK`:LOGICAL; `DIM`: INTEGER (optional)	Returns `TRUE` if any value in `MASK` is `TRUE`. If specified, `DIM` identifies the number of values in `MASK` to check.
ASIN	`Asin(X)`	`X`: REAL	Returns the arcsine of `X`.
ASSOCIATED	`Associated(PTR[,TARGET])`	`PTR`: POINTER; `TARGET`: POINTER, or `TARGET` of same type as `PTR` (optional)	Returns a scalar value of type LOGICAL*4 if `PTR` is associated with any target or the specific `TARGET` specified.
ATAN	`Atan(X)`	`X`: REAL	Returns the arctangent of `X`.
ATAN2	`Atan2(X,Y)`	`X,Y`:REAL	Returns the arctangent of (`X` plus `Y`), where `Y` represents the imaginary part of `X`.
BESJ0	`BesjO(X)`	`X`:REAL	Returns the Bessel function of the first kind of order 0 of `X`.
BESJ1	`Besj1(X)`	`X`:REAL	Returns the Bessel function of the first kind of order 1 of `X`.

Table 3-1. *Intrinsic Functions and Extensions in gfortran (Continued)*

Extension	Invoked As	Parameters	Description
BESJN	`Besjn(X)`	`X`:REAL	Returns the Bessel function of the first kind of order N of `X`.
BESY0	`Besy0(X)`	`X`: REAL	Returns the Bessel function of the second kind of order 0 of `X`.
BESY1	`Besy1(X)`	`X`: REAL	Returns the Bessel function of the second kind of order 1 of `X`.
BESYN	`Besyn(X)`	`X`: REAL	Returns the Bessel function of the second kind of order N of `X`.
BIT_SIZE	`Bit_size(I)`	`I`: INTEGER	Returns the number of bits (integer precision plus sign bit) of `I`.
BTEST	`Btest(I,POS)`	`I`: INTEGER; `POS`: INTEGER	Returns `LOGICAL TRUE` if bit `POS` of `I` is set.
CEILING	`Ceiling(X[, KIND])`	`X`: REAL; `KIND`: Integer initialization expression (optional)	Returns the smallest integer that is greater than or equal to `X`, possibly converted based on the `KIND` expression.
CHAR	`Char(I[,KIND])`	`I`: Integer; `KIND`: Integer initialization expression (optional)	Returns that character represented by `I` in the ASCII character set, possibly converted based on the `KIND` expression.
CHDIR	`CALL Chdir(DIR[, STATUS])`	`DIR`: CHARACTER; `STATUS`: INTEGER	Sets the current working directory to be `DIR`. If the `STATUS` argument is supplied, it contains 0 on success or a nonzero error code otherwise.
CMPLX	`Cmplx(X[,Y,KIND])`	`X`: INTEGER, REAL, or COMPLEX ; `Y`: INTEGER or REAL, optional only allowed if `X` is not COMPLEX ;`KIND`: INTEGER (optional)	Returns a complex number where `X` is converted to the real component, optionally converted based on `KIND`. If `Y` is present, `Y` is converted to the imaginary component, which is 0.0 otherwise.
CONJG	`Conjg(Z)`	`Z`: COMPLEX	Returns the complex conjugate of `(COMPLEX(REALPART(Z), -IMAGPART(Z))`.
COS	`Cos(X)`	`X`: REAL or COMPLEX	Returns the cosine of `X` in a value of the same type as its parameter.

Table 3-1. *Intrinsic Functions and Extensions in gfortran (Continued)*

Extension	Invoked As	Parameters	Description
COSH	`Cosh(X)`	`X`: REAL	Returns the hyperbolic cosine of `X`.
COUNT	`Count(MASK[,DIM])`	`MASK`: LOGICAL ; `DIM`: INTEGER (optional)	Returns an integer holding the number of `TRUE` elements of `MASK` along the dimension of `DIM`. `DIM` is 1 if not specified.
CPU_TIME	`CALL CPU_Time(Seconds)`	`Seconds`: REAL	Returns the current elapsed amount of CPU time used by the calling process in `Seconds`.
CSHIFT	`Cshift(ARRAY, SHIFT[,DIM])`	`ARRAY`: Any Type; `SHIFT`: INTEGER; `DIM`: INTEGER (optional)	Performs a circular shift on elements of `ARRAY` along the dimension of `DIM`, which is 1 if not specified, and returns an array of the same type as `ARRAY`.
DATE_AND_TIME	`CALL DATE_AND_TIME([DATE, TIME, ZONE, VALUES])`	`DATE`: CHARACTER(8) or larger; `TIME`: CHARACTER(10) or larger; `ZONE`: CHARACTER(5) or larger; `VALUES`: INTEGER(8)	Returns the specified information from the system clock. `DATE` has the form ccyymmdd. `TIME` has the form hhmmss.sss. `ZONE` has the form (+–)hhmm, representing the difference with respect to Coordinated Universal Time (UTC). `VALUES` contains the following information: `VALUES(1)`: year, `VALUES(2)`: month, `VALUES(3)`: day of the month, `VALUES(4)`: difference with UTC in minutes, `VALUES(5)`: hour of the day, `VALUES(6)`: minutes of the hour, `VALUES(7)`: seconds of the minute, and `VALUES(8)`: milliseconds of the second.
DBLE	`Dble(X)`	`X`: INTEGER, REAL, or COMPLEX	Returns its argument, converted into a double precision real value.
DCMPLX	`Dcmplx(X[,Y])`	X: INTEGER, REAL or COMPLEX; `Y`: INTEGER or REAL (optional)	Returns a double complex number where `X` is converted to the real component. If `Y` is present it is converted to the imaginary component, which is 0.0 otherwise. `Y` must not be present if `X` is complex.

Table 3-1. *Intrinsic Functions and Extensions in gfortran (Continued)*

Extension	Invoked As	Parameters	Description
DIGITS	`Digits(X)`	`X`: INTEGER or REAL	Returns the number of significant digits of the internal representation of `X`.
DIM	`Dim(X,Y)`	`X`: INTEGER or REAL; `Y`: INTEGER or REAL (same type and kind as `X`)	Returns the difference `X-Y` if the result is positive; otherwise returns zero.
DOT_PRODUCT	`Dot_Product(X,Y)`	`X`, `Y`: Any numeric or LOGICAL	Returns the dot product multiplication of the vectors `X` and `Y`. If the arguments are numeric, the return value is a numeric of the same type. If the arguments are LOGICAL, the return value is a LOGICAL.
DPROD	`Dprod(X,Y)`	`X`, `Y`: REAL	Returns the product `X*Y` as a REAL value.
DREAL	`Dreal(Z)`	`Z`: COMPLEX	Returns the real part of the complex number passed as an argument.
DTIME	`CALL DTIME(ARRAY,RESULT)`	`ARRAY`: REAL, DIMENSION(2); `RESULT`: REAL	Returns the number of seconds of runtime since the start of the process's execution in `RESULT`. Subsequent calls return the elapsed runtime since the last call.
EOSHIFT	`Eoshift(A, SHIFT[,BOUNDARY, DIM])`	`ARRAY`: Any type; `SHIFT`: INTEGER (optional); `BOUNDARY`: Same type as `ARRAY` (optional); `DIM`: INTEGER (optional)	Returns an end-off shift of the elements of `ARRAY` along the dimension of `DIM`, which is 1 if not specified.
EPSILON	`Epsilon(X)`	`X`: REAL	Returns a nearly negligible number relative to 1.
ERF	`Erf(X)`	`X`: REAL	Returns the error function of `X`.
ERFC	`Erfc(X)`	`X`: REAL	Returns the complementary error function of `X`.
ETIME	`CALL ETIME(ARRAY,RESULT)`	`ARRAY`: REAL, DIMENSION(2); `RESULT`: REAL	Returns the number of seconds of execution time since the start of the process's execution in `RESULT`. Subsequent calls return the elapsed execution time since the last call.

Table 3-1. *Intrinsic Functions and Extensions in gfortran (Continued)*

Extension	Invoked As	Parameters	Description
EXIT	`CALL EXIT([STATUS])`	`STATUS`: INTEGER (optional)	Causes immediate termination of the program with the specified `STATUS`, if present.
EXP	`Exp(X)`	`X`: REAL or COMPLEX	Returns the base e exponential of `X`.
EXPONENT	`Exponent(X)`	`X`: REAL or COMPLEX	Returns the base e exponential of `X`.
FLOOR	`Floor(X[,KIND])`	`X`: REAL; `KIND`: integer initialization expression (optional)	Returns the greatest integer less than or equal to `X`, possibly converted as specified by `KIND`.
FLUSH	`CALL Flush(UNIT)`	`UNIT`: INTEGER (optional)	Flushes Fortran unit(s) currently open for output. Without the optional argument, all such units are flushed, otherwise just the unit specified by `UNIT`.
FNUM	`Fnum(UNIT)`	`UNIT`: INTEGER	Returns the POSIX file descriptor number corresponding to the open Fortran I/O unit `UNIT`.
GERROR	`CALL GError(MESSAGE)`	`MESSAGE`: CHARACTER	Returns the system error message corresponding to the last system error (as returned by the C `errno()` function).
GETARG	`CALL GetArg(POS, VALUE)`	`POS`: INTEGER; `VALUE`: CHARACTER	Sets `VALUE` to the `POS`-th command-line argument (or to all blanks if there are fewer than `VALUE` command-line arguments).
GET_COMMAND_ARGUMENT	`CALL Get_Command_Argument (POS, VALUE)`	`POS`: INTEGER; `VALUE`: CHARACTER	Sets `VALUE` to the `POS`-th command-line argument (or to all blanks if there are fewer than `VALUE` command-line arguments).
GET_COMMAND	`CALL Get_Command(VALUE)`	`VALUE`: CHARACTER	Sets `VALUE` to the entire command-line with which the current program was called.
GETCWD	`CALL GetCWD(NAME [, STATUS])`	`NAME`: CHARACTER; `STATUS`: INTEGER (optional)	Places the current working directory in `NAME`. If the `STATUS` argument is supplied, it contains 0 success or a nonzero error code upon return.

Table 3-1. *Intrinsic Functions and Extensions in gfortran (Continued)*

Extension	Invoked As	Parameters	Description
GETENV	`CALL GETENV (NAME, VALUE)`	`NAME,VALUE`: CHARACTER	An F77 intrinsic that retrieves the `VALUE` of the environment specified in `NAME` and returns it in `VALUE`.
GET_ENVIRONMENT_VARIABLE	`CALL GET_ENVIRONMENT_VARIABLE (NAME, VALUE[, LENGTH, STATUS, TRIM_NAME])`	`NAME`, `VALUE`: CHARACTER; `LENGTH`, `STATUS`: INTEGER (optional); `TRIM_NAME`: LOGICAL (optional)	A Fortran 2003 intrinsic that retrieves the `VALUE` of the environment variable specified in `NAME` and returns it in `VALUE`. The length of the `VALUE` is returned in `LENGTH`, and the status of the environment variable lookup is returned in `STATUS`. If `TRIM_NAME` is `TRUE`, trailing spaces are removed from the return `VALUE`.
GETLOG	`CALL GetLog(LOGIN)`	`LOGIN`: CHARACTER; scalar	Returns the login name under which the current process is running in `LOGIN`.
GETXID	`GetXId()`	NONE	Returns the user ID under which the current process is executing.
HOSTNM	`CALL HostNm(NAME, STATUS)`	`NAME`: CHARACTER; `STATUS`: INTEGER (optional)	Fills NAME with the system's host name returned by `gethostname(2)`. If the `STATUS` argument is supplied, it contains 0 on success or a nonzero error code upon return.
IARGC	`IARGC()`	NONE	Returns the number of command-line arguments passed to the program.
IERRNO	`IErrNo()`	NONE	Returns the last system error number (corresponding to the C errno).
ISHFTC	`IShftC(I, SHIFT, SIZE)`	`I`, `SHIFT`, `SIZE`: INTEGER	The rightmost `SIZE` bits of the argument `I` are shifted circularly by `SHIFT` places. The bits shifted out of one end are shifted into the opposite end.
KILL	`CALL Kill(PID, SIGNAL[, STATUS])`	`PID`, `SIGNAL`: INTEGER; `STATUS`: INTEGER (optional)	Sends the signal specified by `SIGNAL` to the process `PID`. If the `STATUS` argument is supplied, it contains 0 on success or a nonzero error code upon return.

Table 3-1. *Intrinsic Functions and Extensions in gfortran (Continued)*

Extension	Invoked As	Parameters	Description
LINK	`CALL Link(PATH1, PATH2, STATUS)`	`PATH1, PATH2`: CHARACTER; `STATUS`: INTEGER (optional)	Makes a (hard) link from file `PATH1` to `PATH2`. If the STATUS argument is supplied, it contains 0 on success or a nonzero error code upon return.
LOG	`Log(X)`	`X`: REAL or COMPLEX	Returns the logarithm of `X`.
LOG10	`Log10(X)`	`X`: REAL or COMPLEX	Returns the Base 10 logarithm of `X`.
MOD	`Mod(A, P)`	`A,P`: INTEGER or REAL	Returns the remainder calculated as `A - (INT(A / P) * P)`. `P` must not be zero.
MVBITS	`CALL MvBits(FROM, FROMPOS, LEN, TO, TOPOS)`	`FROM, FROMPOS, LEN, TO, TOPOS`: INTEGER	Moves `LEN` bits from positions `FROMPOS` through "`FROMPOS+LEN-1`" of `FROM` to positions `TOPOS` through "`FROMPOS+LEN-1`" of `TO`. The portion of argument `TO` that is not affected by the movement of bits is unchanged.
PERROR	`CALL PError(STRING)`	`STRING`: CHARACTER	Prints a newline-terminated error message corresponding to the last system error. This is prefixed by `STRING`, a colon, and a space.
RAND	`Rand(FLAG)`	`FLAG`: INTEGER (optional)	Returns a uniform quasi-random number between 0 and 1. If `FLAG` is 0, the next number in sequence is returned; if `FLAG` is 1, the generator is restarted. Passing any other value for `FLAG` causes the function to use `FLAG` as the seed for a newly generated random number.
REAL	`Real(X[,KIND])`	`X`: INTEGER, REAL, or COMPLEX; `KIND`: Optional scalar integer initialization expression	Returns a REAL value corresponding to its argument.
RENAME	`CALL Rename(PATH1, PATH2, STATUS)`	`PATH1,PATH2`: CHARACTER; `STATUS`: INTEGER (optional)	Renames the file `PATH1` to `PATH2`. If `STATUS` is supplied, it contains 0 on success or a nonzero error code on error.

Table 3-1. *Intrinsic Functions and Extensions in gfortran (Continued)*

Extension	Invoked As	Parameters	Description
SIGN	`Sign(A, B)`	`A`,`B`: INTEGER or REAL	Returns "`ABS(A)*s`", where `s` is +1 if "`B.GE.0`," –1 otherwise.
SIN	`Sin(X)`	`X`: REAL or COMPLEX	Returns the sine of `X`.
SINH	`Sinh(X)`	`X`: REAL	Returns the hyperbolic sine of `X`.
SLEEP	`CALL Sleep(SECONDS)`	`SECONDS`: INTEGER	Causes the process to pause for `SECONDS` seconds.
SQRT	`Sqrt(X)`	`X`: REAL or COMPLEX	Returns the square root of `X` in the same type as passed as a parameter.
STAT	`CALL Stat(FILE, SARRAY[, STATUS])`	`FILE`: CHARACTER; `SARRAY`: INTEGER DIMENSION(13); `STATUS`: INTEGER (Optional)	Obtains data about the file `FILE` and places them in the array `SARRAY`. The values in this array are extracted from the stat structure as returned by the Linux fstat(2) function. If the `STATUS` argument is supplied, it contains 0 on success or a nonzero error code upon error.
SYMLINK	`CALL SymLnk(PATH1, PATH2[, STATUS])`	`PATH1`: CHARACTER; `PATH2`: CHARACTER; `STATUS`: INTEGER (optional)	Makes a symbolic link from file `PATH1` to `PATH2`. If the `STATUS` argument is supplied, it contains 0 on success or a nonzero error code.
SYSTEM	`CALL System(COMMAND [, STATUS])`	`COMMAND`: CHARACTER; `STATUS`: INTEGER (optional)	Passes the command `COMMAND` to a shell (see system(3)). If argument `STATUS` is present, it contains the value returned by `system(3)`, which is 0 if the shell command succeeds. Note that the shell used to invoke the command is environment-dependent.
SYSTEM_CLOCK	`CALL System_Clock(COUNT [, RATE, MAX])`	`COUNT`: INTEGER; `RATE`: INTEGER (optional); `MAX`: INTEGER (optional)	Returns the current value of the system clock in `COUNT`. If provided, `RATE` is the number of clock ticks per second and `MAX` is the maximum value this can take.
TAN	`Tan(X)`	`X`: REAL	Returns the tangent of `X`.
TANH	`Tanh(X)`	`X`: REAL	Returns the hyperbolic tangent of `X`.

Table 3-1. *Intrinsic Functions and Extensions in gfortran (Continued)*

Extension	Invoked As	Parameters	Description
TIME	`Time()`	NONE	Returns the current time encoded as an INTEGER.
UMASK	`CALL UMask(MASK[, Old])`	`MASK`: INTEGER; `OLD`: INTEGER (optional)	Sets the file creation mask to `MASK` and returns the old value in argument `OLD` if it is supplied. See `umask(2)`.
UNLINK	`CALL Unlink(FILE [, STATUS])`	`FILE`: CHARACTER; `STATUS`: INTEGER (optional)	Unlink the file `FILE`. If the `STATUS` argument is supplied, it contains 0 on success or a nonzero error code.

Classic GNU Fortran: The g77 Compiler

As mentioned at the beginning of this chapter, gfortran is being developed as a replacement for g77. However g77 is still stable, available, and widely used. Ironically, one of the reasons that developer Burley gives for not continuing active g77 development (see `http://world.std.com/~burley/g77-why.html`) is that g77 is so stable that few of its users are interested in new features or significant enhancements to the existing code base. "Don't fix it if it isn't broken" is probably one of the best compliments that any software developer can receive, especially in something as complex as a compiler.

■ **Note** This is probably the right place for a semantic tip of the hat to James Craig Burley, especially, and to everyone else who contributed to g77. GCC's g77 compiler helped thousands of people continue to use and enhance FORTRAN 77 code, with no cost to the user. The g77 compiler is a stunning and impressive example of the power and benefits of open source software. Thanks, everyone!

Why Use g77?

At the time this book was written, the current gfortran documentation recommended that developers continue to use g77 for compiling Fortran code that strictly conforms to either the FORTRAN 77 standard or only takes advantage of extensions provided by the GNU FORTRAN 77 language.

GNU FORTRAN 77, g77, is still available—it is open source, after all. The last version of GCC that included g77 was GCC 3.4.5. For more information about this release, see `http://gcc.gnu.org/gcc-3.4/`. The complete documentation for the 3.4.5 release of g77 is available at `http://gcc.gnu.org/onlinedocs/gcc-3.4.5/g77/`. For specific information about the status of g77 in GCC 3.4.5, see `http://gcc.gnu.org/onlinedocs/gcc-3.4.5/g77/News.html`. For general information about g77, see `http://world.std.com/~burley/g77.html`. You can even get a Mac OS X version of g77 from `http://hpc.sourceforge.net/`.

Differences Between g77 and gfortran Conventions

One of the primary differences between g77 and gfortran is the lack of case-sensitivity in gfortran. g77 provided more than 20 command-line options for dealing with uppercase and lowercase characters in Fortran input—in the names of intrinsic functions, in symbol names, and so on. (See `http://gcc.gnu.org/onlinedocs/gcc-3.4.5/g77/Case-Sensitivity.html` for details and entertainment.)

gfortran eliminates this confusion by ignoring case, which means your Fortran programs that have separate subroutines named *Asub*, *aSub*, and *asuB* will no longer link correctly. You should be ashamed of yourself anyway. Please update your code to use more meaningful and distinctive names, and destroy all of your old case-reliant source code.

As discussed in the section "Compiling Fortran Applications with gfortran," code compiled with g77 relied on the `-ff90` and `-ff95` options to tell g77 to accept non-FORTRAN 77 capabilities, such as free-form input, some C language conventions, different intrinsics and extensions, and so on. These options are no longer necessary (or supported) in gfortran. Instead, gfortran uses the extension of your input program to identify the version of Fortran that your code conforms to. The following extensions have special meaning to the gfortran compiler:

- `.f`: Interpreted as FORTRAN 77 code
- `.f90`: Interpreted as Fortran 90 code
- `.f95`: Interpreted as Fortran 95 code

If you don't feel like renaming your input files but, for example, need a program with the `.f` extension to be interpreted as Fortran 95 code, you can specify the `-ffree-form` option on the gfortran command line to override FORTRAN 77 format conventions.

The g77 and gfortran compilers also use different conventions for creating the entity names used in object modules and symbol tables. Like the f2c program, the g77 compiler appends an underscore to the name of every entity found in the input source file whose name does not already end with an underscore. This is done to ensure compatibility with code produced by many old-school Unix Fortran compilers. However, the gfortran compiler does not append an underscore by default—you must explicitly specify the `-funderscoring` option to cause gfortran to behave in the same way as f2c and g77.

Of course, just because symbol, function, and subroutine names match, it does not mean that the interface implemented by gfortran for an external name matches the interface with the same name as implemented by some other language or Fortran variant. Getting code produced by gfortran to link to code produced by another compiler is only the first step in the integration process. Getting the code generated by both compilers to agree on issues other than naming can require significant effort and can be harder to resolve because the linker can't identify internal inconsistencies beyond name and parameter disagreements.

Finally, g77 offered a variety of options for interoperability with other Fortran compilers. The most commonly used of these was the `-fvxt` option, which provided interoperability with Digital VAX FORTRAN (formerly available from Digital Equipment Corporation—DEC—and now presumably available from Compaq, which bought DEC). Using this option caused constructs that were shared between VAX FORTRAN and g77 to be interpreted as they would be in VAX FORTRAN rather than using the canonical g77 interpretation. This option is not supported by gfortran, so you may have to make code changes if you relied on this option and your Fortran code no longer compiles or links correctly.

Other options provided by g77 for interoperability, but which are no longer supported, are the legendary `-ugly` options. These were `-fugly-args`, `-fugly-assumed`, `-fugly-comma`, `-fugly-complex`, `-fugly-init`, `-fugly-logint`, and `-fugly-assign`. These options are no longer available—sorry, but you'll finally have to get around to changing your ancient source code to work around the compatibility issues these options addressed. Or, as discussed at the beginning of this section, you can just continue to use g77 forever.

Alternatives to gfortran and g77

This may initially seem to be an odd section in a book on GCC, but the fact that GCC has changed Fortran compilers may leave some people in the lurch. The previous section explained how to obtain

and keep using g77. This section discusses alternative Fortran utilities and compilers that may help you work around any problems that you encounter in gfortran. You should always report bugs (so they can be fixed), but filing a bug may be small consolation if you can't compile code that you actually need to use right now.

The f2c Fortran-to-C Conversion Utility

The f2c program, a Fortran-to-C converter whose source code is freely available, was much of the inspiration for the input handling and code parsing routines of g77. The output of f2c uses the same conventions as g77 (and gfortran, if the `-ff2c` option is specified or active). Code translated from Fortran to C using f2c and compiled with a compiler such as GCC can be linked and used with Fortran code compiled by g77 and gfortran. The f2c program supports ANSI FORTRAN 77 plus some popular extensions.

The latest f2c source lives at `http://www.netlib.org/f2c` and at Bell Labs at `http://cm.bell-labs.com/netlib/f2c/`, but not in a very useful fashion. To retrieve the latest f2c source, get the GCC 3.4.5 release, unpack it, and use the `download_f2c` script in `gcc-3.4.5/contrib` to retrieve and unpack the latest f2c source code into a directory whose name identifies the current version and last update time. At the time this book was written, the latest version was f2c-20050928.

The calling conventions used by g77 (originally implemented in f2c) require functions that return type default `REAL` to actually return the C type double, and functions that return type `COMPLEX` to return the values via an extra argument in the calling sequence that points to where to store the return value. Under the default GNU calling conventions, such functions simply return their results as they would in GNU C—default `REAL` functions return the C type float, and `COMPLEX` functions return the GNU C type complex. For this reason, it is not a good idea to mix Fortran code translated or compiled using f2c or g77 code with code compiled by gfortran using the `-fno-f2c` option. Calling `COMPLEX` or default `REAL` functions between program parts that were compiled with different calling conventions will break at execution time.

The g95 Fortran Compiler

The g95 Fortran compiler is another well-known open source compiler that is available for Linux, as well as for a number of other platforms. The latest versions of this compiler are available at `http://g95.sourceforge.net/`. The authors of g95 recently announced that they had reached the point where there were no known outstanding bugs in the compiler, which is quite a feat.

Intel's Fortran Compiler

Intel offers a Fortran compiler for Linux. This compiler has a good reputation for generating high-performance code. For more information, see `http://www.intel.com/cd/software/products/asmo-na/eng/compilers/flin/index.htm`.

Additional Sources of Information

Like all GNU projects, a vast amount of information about gfortran is available from a variety of sources on the Web. Some of the best sources of information specific to gfortran are the following:

- The Fortran mailing list at `http://gcc.gnu.org/lists.html` is a very active list where you can ask questions, discuss bugs, and participate in helping set the future direction of gfortran. Since the GNU folks host a large number of mailing lists, you'll have to scroll down quite a way on this page to find the subscription information for this list.
- If you don't want to subscribe to the Fortran mailing list, you can read archived collections of old posts at `http://gcc.gnu.org/ml/fortran`. These are archived on a monthly basis and provide a great resource for determining whether any gfortran problem you may be experiencing is one that someone has already experienced and resolved.
- The list of current open bugs in gfortran is available in the Fortran section of `http://gcc.gnu.org/onlinedocs/gcc-3.4.5/g77/Trouble.html`. Unfortunately, at the time this book was written, this page still referred to g77, but it's bound to catch up with the actual software one of these days.

In general, the gfortran developers welcome any feedback on user experience with gfortran via e-mail to `fortran@gcc.gnu.org`.

In addition to these GNU Fortran resources, a variety of more general Fortran resources is available on the Web. (Big surprise!) Some of my favorites are as follows:

- Michael Metcalf's page on Fortran 90/95/HPF (High Performance Fortran), at `http://www.fortran.com/metcalf.htm`, is a great general reference for information about modern and available Fortran compilers. It also provides titles of useful books on Fortran, other useful Web sites, and so on.
- The Fortran open directory entry at `http://dmoz.org/Computers/Programming/Languages/Fortran/` is another great general-purpose reference site for information about Fortran, compilers, and companies that still specialize in Fortran tools and consulting.
- The g95 Project's Web site, `http://www.g95.org`, is understandably focused on the g95 Fortran compiler, which was discussed earlier in this chapter.
- The Fortran Company's Web site at `http://www.fortran.com` is similarly focused on its own products, but also provides a number of links to other Fortran compilers, other Fortran-oriented sites on the Web, and so on. Its general information page (`http://www.fortran.com/info.html`) is especially useful.

CHAPTER 4

Using GCC's Java Compiler

This chapter discusses typical usage of the GNU Compiler Collection's Java compiler, gcj, focusing on the command-line options and constructs that are specific to gcj and the additional tools that are provided with gcj.

Java and GCC's Java Compiler

Java is a popular language for a number of reasons, and the inclusion of a Java compiler in the GNU Compiler Collection is a huge step forward for the popularity of Java on Linux systems. This chapter of the book is about using gcj, rather than providing a cheerleading session for Java in general. Hundreds of other books provide the latter. However, it is important to provide a bit of background about Java, and, specifically, about how Java compilers work, in order to understand what gcj does and how best to take advantage of its capabilities.

Aside from inherent features of the language, such as its object orientation, security, and network awareness, the biggest reason for Java's popularity is the portability of Java applications. Java source code is typically precompiled into system-independent bytecode which can subsequently be executed by any Java Virtual Machine (JVM) that is running on a specific target platform. However, this portability and subsequent interpretation has a performance downside—interpretation takes time. Many JVMs therefore include what are known as Just-in-Time (JIT) or HotSpot compilers. These compilers are internal to the JVM and look for frequently executed portions of the bytecode that they then precompile into optimized machine-specific object code at execution time. Commercial examples of Linux JVMs that use the JIT approach on Intel platforms include Sun's HotSpot Client and Server JVMs (`http://java.sun.com/j2se`), BEA's WebLogic JRockit (`http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/jrockit`), and IBM's Java 2 Runtime Environment (`http://www-128.ibm.com/developerworks/java/jdk/linux/`). The best-known open source JVMs using the JIT approach under Linux are Blackdown's Java Platform 2 for Linux (`http://www.blackdown.org/`), Kaffe (`http://www.kaffe.org/`), and the SableVM (`http://sablevm.org/`), which are also available for non-Intel platforms, thanks to the nature of open source.

The other common approach to improving Java performance on a specific platform is known as Ahead-of-Time (AOT) compilation, which is essentially the traditional approach to compilation tweaked slightly for the Java environment. AOT compilers can compile Java source code into traditional Java bytecode or into platform-specific object code, and can also compile existing bytecode into platform-specific object code. GCC's gcj Java compiler is an AOT compiler that comes with its own runtime, libgcj, which provides a bytecode interpreter (a built-in version of the GNU interpreter for Java, gij), a set of core class libraries, and a garbage collector. The core class libraries are based on the GNU Classpath project (`http://www.gnu.org/software/classpath/`) and are in the process of merging with that project. The built-in interpreter is provided to handle Java code that is not compiled to object code ahead of time. Java bytecode typically provides external components or class libraries,

which can also be applications that you simply want to deliver in traditional Java bytecode. Another popular AOT compiler for Linux systems is Excelsior JET (`http://www.excelsior-usa.com/jet.html`), though this is not an open source product/project.

Basic gcj Compiler Usage

Appendix A discusses the options that are common to all of the GCC compilers, and how to customize various portions of the compilation process. However, I'm not a big fan of making people jump around in a book just to simplify its organization. For that reason, this section provides a quick refresher of basic GCC compiler usage as it applies to the gcj Java compiler. For detailed information, see Appendix A. If you are new to gcj or the GCC compilers in general and just want to get started quickly, you're in the right place.

The gcj compiler accepts both single-letter options, such as `-o`, and multiletter options, such as `--classpath`. Because it accepts both types of options you cannot group multiple single-letter options together as you may be used to doing in many GNU and Unix/Linux programs. For example, the multiletter option `-pg` is not the same as the two single-letter options `-p -g`. The `-pg` option creates extra code in the final binary that outputs profile information for the GNU code profiler, gprof. On the other hand, the `-p -g` options generate extra code in the resulting binary that produces profiling information for use by the prof code profiler (`-p`) and causes gcj to generate debugging information using the operating system's normal format (`-g`).

Despite its sensitivity to the grouping of multiple single-letter options, you are generally free to mix the order of options and compiler arguments on the gcc command line—that is, invoking gcj as

```
gcj -pg -fno-strength-reduce -g myprog.java -o myprog --main=Hello
```

has the same result as

```
gcj myprog.java --main=Hello -o myprog -g -fno-strength-reduce -pg
```

I say that you are generally free to mix the order of options and compiler arguments because, in most cases, the order of options and their arguments does not matter. In some situations, order does matter, if you use several options of the same kind. For example, the `-I` option specifies an additional directory or directories to search for jar (Java Archive) files. So, if you specify `-I` several times, gcj searches the listed directories in the order specified.

The following is the classic C language hello, world application, written in Java as a trivial example for this section:

```
class Hello
{
        public static void main(String args[])
        {
           System.out.println("Hello World!");
        }
}
```

After saving this to a file named hello.java, it is easy to compile it to produce an executable using gcj—just invoke gcj, passing the name of the source file as the argument and using the `--main` option to identify the name of the class containing the `main` method that should be called when the application executes.

```
$ gcj --main=Hello hello.java
$ ls -l
```

```
-rwxr-xr-x    1 wvh    users      14256 Oct   5 16:17 a.out
-rw-r--r--    1 wvh    users      134 Oct  5 16:17 hello.java
```

By default, the result on Linux and Unix systems is an executable file named a.out in the current directory, which you execute by typing **./a.out**. On Cygwin systems, you will wind up with a file named a.exe that you can execute by typing either **./a** or **./a.exe**. Running the a.out file on a Linux system produces the familiar output, as shown in the following example:

```
$ ./a.out
```

```
Hello World!
```

ERROR MESSAGES FOR MAIN METHODS

An error message like the following is a common message that you may see when attempting to compile a Java application:

```
$ gcj hello.java
```

```
/usr/lib64/gcc/x86_64-suse-linux/4.0.2/../../../../lib64/crt1.o: ➥
     In function `_start':
../sysdeps/x86_64/elf/start.S:109: undefined reference to `main'
collect2: ld returned 1 exit status
```

This message simply means that you've forgotten to specify the `main` method using the `--main` option, and is analogous to the following type of message from an actual Java Virtual Machine:

```
$ java Hello.java
```

```
Exception in thread "main" java.lang.NoClassDefFoundError: Hello
```

To resolve the gcj compilation message, simply specify the `main` method using the `--main` command-line option. To resolve the equivalent problem in a Java Virtual Machine, you can simply use gcj to generate a class file that contains the appropriate `main` method information and then run that using the Java interpreter, as in the following example:

```
$ gcj -C hello.java
$ java Hello
```

```
Hello World!
```

To define the name of the output executable that gcj produces, use the `-o` option, as illustrated in the following example:

```
$ gcj hello.java --main=Hello -o runme
$ ls -l
```

```
-rw-r--r--    1 wvh    users       134 Oct  5 16:17 hello.java
-rwxr-xr-x    1 wvh    users     14256 Oct  5 16:28 runme
```

If you are compiling multiple source files using gcj, you can simply specify them all on the gcj command line, as in the following example, which leaves the compiled and linked executable in the file named showdate:

```
$ gcj showdate.java helper.java --main=ShowDate -o showdate
```

If you want to compile these files incrementally and eventually link them into a binary, you can use the `-c` option to halt compilation after producing an object file, and then link the object files together, as in the following example:

```
$ gcj -c showdate.java
$ gcj -c helper.java
$ gcj showdate.o helper.o --main=ShowDate –o showdate
$ ls -l
```

```
total 124
-rw-r--r--    1 wvh    users          210 Oct  5 12:42 helper.java
-rw-r--r--    1 wvh    users         1104 Oct  5 13:50 helper.o
-rwxr-xr-x    1 wvh    users        13891 Oct  5 13:51 showdate
-rw-r--r--    1 wvh    users          208 Oct  5 12:44 showdate.java
-rw-r--r--    1 wvh    users         1008 Oct  5 13:50 showdate.o
```

Because gcj also supports compilation into bytecode class files, you could also achieve the same thing by first using gcj's `–c` option, which tells the compiler to generate class files and then exit, and then using the `–o` and `--main` options together to generate object code and link the files with the correct `main` method, as in the following example:

```
$ gcj -C showdate.java
$ gcj -C helper.java
$ gcj ShowDate.class Helper.class --main=ShowDate –o showdate
$ ls -l
```

```
total 124
-rw-r--r--    1 wvh    users          210 Oct  5 12:42 helper.java
-rw-r--r--    1 wvh    users          310 Oct  5 13:50 Helper.class
-rwxr-xr-x    1 wvh    users        13891 Oct  5 13:51 showdate
-rw-r--r--    1 wvh    users          208 Oct  5 12:44 showdate.java
-rw-r--r--    1 wvh    users          780 Oct  5 13:50 ShowDate.class
```

Using bytecode as your intermediate format is most useful either when this is your target output format, or when you are compiling multiple Java source files and incrementally testing the classes that they contain using the gij.

■ **Note** All of the GCC compilers "do the right thing" based on the extensions of the files provided on any GCC command line. Mapping file extensions to actions (for example, understanding that files with `.o` extensions only need to be linked) is done via the GCC specs file. Prior to GCC version 4, the specs file was a stand-alone text file that could be modified using a text editor; with GCC 4 and later, the specs file is built-in and must be extracted before it can be modified. For more information about working with the specs file, see "Customizing GCC Using Spec Strings" in Appendix A.

It should be easy to see that a project consisting of more than a few source code files would quickly become exceedingly tedious to compile from the command line, especially after you start adding search directories, complex dependencies, optimizations, and other gcj options. Most Java developers use the Apache project's Ant utility (`http://ant.apache.org/`) to solve this problem rather than the traditional make utility. This is only natural because Ant is itself based on Java and executes Java objects to perform the various tasks required to resolve dependencies in an Ant configuration file. Ant's build.xml configuration files use a modern, easily-understood XML format rather than the admittedly voodoolike format of complex Makefiles. Though many people no longer manually write Makefiles thanks to autoconf and automake (see Chapter 7), not even the most hard-core make fan could claim that it is harder to read or write build.xml files than autoconf-generated Makefiles. Like discussing make, discussing Ant is also outside the scope of this book, but Ant's online documentation is excellent, as are various books on Ant, such as *Ant: The Definitive Guide, Second Edition,* Steve Holzner (O'Reilly, 2005. ISBN: 0-596-00609-8).

Demonstrating gcj, javac, and JVM Compatibility

While gcj can clearly compile Java applications into object code that your system can execute with the help of libgcj.jar, gcj can also easily be used to compile existing class and jar files into executables that your system can execute. While this should be no surprise given the basic design of Java and the whole idea of system-independent bytecode, it's still useful to demonstrate this.

To do so, I'll use a Java version of the standard Fibonacci sequence application used in all of the other precompiler chapters in this book. The sample code for this application, stored in the file Fibonacci.java, is the following:

```
import java.io.PrintStream;

public class Fibonacci {

    private int sequence = 0;

    private int calcFibonacci(int n) {
         if(n <= 1)
             return n;
         else
             return calcFibonacci(n - 1) + calcFibonacci(n - 2);
    }

    public void calculate(PrintStream out) {
        int j;
        for (j = 0 ; j < sequence ; j++)
            out.print(calcFibonacci(j) + " ");
        out.println();
    }
```

```
    public Fibonacci(int i) {
        sequence = i;
    }

    public static void main(String[] argv) {
        if ( argv.length == 0) {
            System.out.println("Usage: fibonacci num-of-sequence-values-to-print");
            System.exit(0);
        } else
            new Fibonacci(new Integer(argv[0]).intValue()).calculate(System.out);
    }
}
```

The working directory initially holds just this source code (and a copy of the hello, world application in Java):

```
$ ls -l
```

```
total 8
-rw-r--r--  1 wvh users 544 2006-02-19 09:39 Fibonacci.java
-rw-r--r--  1 wvh users 134 2006-02-19 09:39 hello.java
```

Let's compile these applications using Sun's Java compiler, javac (part of the J2SE that you can freely download from `http://java.sun.com/`), using the `-verbose` option to display the version of the JVM libraries used during compilation:

```
$ javac Fibonacci.java
$ javac -verbose hello.java
```

```
[parsing started hello.java]
[parsing completed 63ms]
[loading /usr/lib/jvm/java…/jre/lib/rt.jar(java/lang/Object.class)]
[loading /usr/lib/jvm/java…/jre/lib/rt.jar(java/lang/String.class)]
[checking Hello]
[loading /usr/lib/jvm/java…/jre/lib/rt.jar(java/lang/System.class)]
[loading /usr/lib/jvm/java…/jre/lib/rt.jar(java/io/PrintStream.class)]
[loading /usr/lib/jvm/java…/jre/lib/rt.jar(java/io/FilterOutputStream.class)]
[loading /usr/lib/jvm/java…/jre/lib/rt.jar(java/io/OutputStream.class)]
[wrote Hello.class]
[total 316ms]
```

At this point, the directory only contains the source code and the class files generated by javac:

```
$ ls -l
```

```
total 20
-rw-r--r--  1 wvh users 985 2006-02-19 09:52 Fibonacci.class
-rw-r--r--  1 wvh users 544 2006-02-19 09:39 Fibonacci.java
-rw-r--r--  1 wvh users 416 2006-02-19 09:52 Hello.class
-rw-r--r--  1 wvh users 134 2006-02-19 09:39 hello.java
```

Now, let's execute these class files using Sun's 1.5.0 JVM:

```
$ java -showversion
```

```
java version "1.5.0_02"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_02-b09)
Java HotSpot(TM) Client VM (build 1.5.0_02-b09, mixed mode, sharing)
….
```

```
$ java Hello
```

```
Hello World!
```

```
$ java Fibonacci 7
```

```
0 1 1 2 3 5 8
```

Now, I'll use gcj to compile the Fibonacci.class file produced by javac, demonstrate that the output is an actual binary, and then execute that object code:

```
$ gcj Fibonacci.class -o fibonacci --main=Fibonacci
$ file fibonacci
```

```
fibonacci: ELF 64-bit LSB executable, AMD x86-64, version 1 (SYSV), ➥
        for GNU/Linux 2.4.1,
        dynamically linked (uses shared libs), not stripped
```

```
$ ./fibonacci 7
```

```
0 1 1 2 3 5 8
```

Just to be completely paranoid, let's try this in reverse, using gcj to generate a class file and then executing that class file using Sun's JVM:

```
$ gcj -C Fibonacci.java
$ java Fibonacci 7
```

```
0 1 1 2 3 5 8
```

As mentioned before, none of this should have come as a surprise, but it's useful to actually demonstrate that gcj does indeed hold to standard class file conventions, which makes it possible to mix and match class files compiled with gcj with existing class files without recompilation.

Filename Extensions for Java Source Files

As mentioned in the previous section, all GCC compilers evaluate filename suffixes to identify the type of compilation that they will perform. Table 4-1 lists the filename suffixes that are relevant to gcj and the type of compilation that gcj performs for each.

Table 4-1. *GCC Filename Suffixes for Java*

Suffix	Operation
`.class`	Java bytecode file.
`.jar`	Archive file containing one or more `.class` files. The archive may or may not be compressed.
`.java`	Java source file.
`.zip`	Compressed archive file containing one or more `.class` files.

A filename specified on the gcj command line with no recognized suffix is considered an object file to be linked. A file in a `.jar` or `.zip` file with no extension is considered to be a Java resource file identifying resources that can be accessed at runtime as `core:/`*`resource-name`* URLs using the core protocol handler.

As with the other GCC compilers, this fixed list of filename suffixes does not mean you are limited to using these suffixes to identify source or object files. As discussed in Appendix A, you can use the `-x` *`lang`* option to identify the language used in one or more input files if you want to use a nonstandard extension. The `lang` argument tells gcj the input language to use.

■ **Note** When any GCC compiler encounters a file with one of the suffixes shown in Table 4-1, it treats the file as a Java file. Nonetheless, other GCC compilers (such as gcc) do not understand the complete chain of dependencies, such as class libraries, that Java programs often require and also do not directly know how to compile Java code. You should therefore always use gcj to invoke GCC's Java compiler directly when compiling Java code rather than using the gcc command and hoping for the best.

Command-Line Options for GCC's Java Compiler

As explained in Appendix A, many command-line options are common to all of the compilers in the GCC suite. Table 4-2 shows the command-line options that are specific to the gcj compiler.

Table 4-2. *gcj Command-Line Options*

Option	Description
`--bootclasspath=`*`PATH`*	Identifies the location of the standard built-in classes, such as `java.lang.String`. See the next section, "Constructing the Java Classpath," for more information about this option and class location.
`-c`	Specifies that all input files are Java source files and compiles each of them into Java bytecode `.class` files. No object code (either `.o` files or a single executable) is produced.

Table 4-2. *gcj Command-Line Options*

Option	Description
`--classpath=`*`PATH`*	Sets the classpath to *PATH*. This does not override the built-in `boot` search path. See the next section, "Constructing the Java Classpath," for more information about this option and the classpath in general.
`--CLASSPATH=`*`PATH`*	Provides a deprecated synonym for the `--classpath` option.
`-d` *`OUTPUTDIR`*	Provides a synonym for the `--output-class-dir` option, which exists for compatibility with the options used by Sun's javac Java compiler.
`-D`*`name`*`[=value]`	Defines a system property named *name* with the value *value*—if no value is specified, the default value of name is the empty string. Any system properties defined using this option can be retrieved at runtime using the `java.lang.System.getProperty` method. This option can only be used when linking—in other words, in conjunction with the `--main` or `-lgij` options.
`--encoding=`*`NAME`*	Enables you to manually specify the encoding used in one or more Java source files. Ordinarily, gcj first tries to use the default encoding specified by your current locale, falling back to UTF-8 Unicode encoding if your system does not have sufficient locale support. The gcj compiler uses UTF-8 internally, so this is always safe to use in your source files.
`--extdirs=`*`PATH`*	Identifies one or more additional directories that should be appended to the end of the classpath. See the next section, "Constructing the Java Classpath," for more information about this option and the classpath in general.
`-lgij`	Generates object code that uses the same command-line processing mechanism as the GNU interpreter for Java's gij command to identify the name of the class whose `main` method is the entry point for program execution. This command-line option is an alternative to using the `--main` option when generating an executable and is ignored if the `--main` option is also specified.
`--main=`*`CLASSNAME`*	Identifies the class whose `main` method is the entry point for executing the application. Required when linking a gcj executable.
`--output-class-dir=`*`OUTPUTDIR`*	Specifies the top-level directory to which `.class` output files should be written. Used with the `-c` option. The `.class` files generated by gcj's `-c` option follow the same directory structure as input file. For example, the class file for the input file *foo/*bar.java would be written to `OUTPUTDIR/foo/bar.class`.
`--resource` *`resource-name`*	Tells gcj to compile the contents of the specified file to object code so it may be accessed at runtime with the core protocol handler as `core:/`*`resource-name`*. This option is typically used to compile files with no extension for subsequent inclusion into jar or zip archives.
`-shared`	Tells gcj to create a shared library as its output rather than a standard executable.

■ **Note** By default, gcj generates code with a debugging level of 1 (line number information) rather than using the standard gcc convention of not including debugging information. This is overridden if you specify a particular debugging or optimization level using any `-g` or `-o` option.

The gcj compiler recognizes other options specific to Java warnings, code generation, and optimizations, which are discussed in more detail in the appropriate sections of Appendix A. However, for your convenience, Table 4-3 summarizes the gcj-specific options related to warnings, and Table 4-4 summarizes the gcj-specific options related to code generation and optimization.

Table 4-3. *gcj Warning Option Summary*

Option	Description
`-Wredundant-modifiers`	Causes gcj to generate a warning when a redundant modifier is encountered, such as a public declaration for an interface method.
`-Wextraneous-semicolon`	Causes gcj to generate a warning when an empty statement is encountered.
`-Wno-out-of-date`	Suppresses the warning that gcj generates by default when a source file is newer than its matching class or object file.
`-Wno-deprecated`	Causes gcj to generate a warning if a deprecated class, method, or field is referred to.
`-Wall`	Implies the `-Wredundant-modifiers`, `-Wextraneous-semicolon`, and `-Wunused` options.

Table 4-4. *gcj Code Generation and Optimization Option Summary*

Option	Description
`--disable-assertions[=`*`class-or-package`*`]`	Tells gcj not to embed code for checking assertions in the compiled code. If no class or package is specified, using this option disables assertion code generation for all classes and packages, unless a more specific `--enable-assertions` option is also specified. If a class or package name is specified, this option disables assertion checks within the named class and its inner classes or the named package and any subpackages. By default, assertions are enabled when generating class files (`-c`) or when not optimizing (no `-o` option), and disabled when generating optimized binaries using any `-o` option.
`--enable-assertions[=`*`class-or-package`*`]`	Tells gcj to embed code for checking assertions in the compiled code. As of gcj 4.1, this option is only useful to partially override the effects of a more sweeping `--disable-assertions` option.

Table 4-4. *gcj Code Generation and Optimization Option Summary*

Option	Description
`-findirect-dispatch`	Enables a special binary compatibility ABI that honors the binary compatibility guarantees in the Java Language Specification, causing all dependencies to be looked up at runtime. This allows free mixing of interpreted and compiled code. At the time this book was written, this option could only be used when compiling class files whose native methods are written using the Java Native Interface (JNI).
`-fjni`	Tells gcj that the native methods in the class(es) being compiled are written in the JNI rather than the default Compiled Native Interface (CNI), and causes gcj to generate stubs that will invoke the underlying JNI methods.
`-fno-assert`	Tells gcj not to recognize the `assert` keyword. This option is provided for compatibility with older versions of the language specification.
`--no-bounds-check`	Tells gcj not to automatically embed instructions to check at runtime that array indices are within bounds. By default, gcj automatically embeds instructions to verify that all array indices are within bounds, causing the application to throw an `ArrayIndexOutOfBoundsException` exception if they are not.
`-fno-optimize-static-class-initialization`	Tells gcj not to optimize how static classes are initialized, regardless of the specified optimization level. By default, when the optimization level is `-O2` or greater and object code is being produced, gcj tries to optimize how the runtime initializes static classes the first time that they are used.
`-fno-store-check`	Tells gcj not to automatically embed instructions to check at runtime that objects stored into an array are compatible with the type of that array. With this option, these checks are omitted. By default, gcj automatically embeds instructions to verify that all code that stores an object into an array is assignment-compatible with the type of that array, causing the application to throw an `ArrayStoreException` exception if they are not.

Constructing the Java Classpath

Like all Java compilers, gcj uses the notion of a *classpath* to identify directory locations from which to load classes that are included by reference in files that are being compiled. By default, gcj looks in libgcj.jar, which is a jar archive containing all of the standard classes provided with gcj and whose location is compiled into gcj. The gcj compiler provides the `--bootclasspath`, `--classpath`, and `--extdirs` options to identify alternate locations for class or Java source files that are included by reference, each of which takes an argument that consists of one or more directories. On Linux systems, multiple directories specified as an argument to these options must be separated by a colon; on Windows systems, they must be separated by a semicolon. The gcj compiler also uses the standard GCC `-I` option for this purpose. All of these options interact with the `CLASSPATH` environment variable that is used for the same purpose.

The file/directory search order used by gcj during compilation is the following:

- The compiler searches its shared library path for `.so` files that may contain the requested class. For example, if the class `foo.bar` is requested, gcj will search for shared libraries with the names `lib-foo-bar.so` and `lib-foo.so` (in that order) before proceeding to the rest of the classpath search.
- The compiler next searches any directories specified using the `-I` option.
- If the `--classpath` option is specified, the specified directories are searched in order.
- If the `--classpath` option is not specified and the `CLASSPATH` environment variable is set, the directories specified in the `CLASSPATH` are searched in order.
- The current directory is searched.
- Any directory specified using the `--bootclasspath` option is searched.
- If the `--bootclasspath` option was not specified, the built-in system class archive, libgcj.jar is searched.
- If the `--extdirs` option was specified, the specified directory is searched in order.
- If the `--extdirs` option was not specified, the built-in directory in *`install-location`*`/share/java/ext` is searched.

■ **Note** The default class file for the class `java.lang.Object` (stored in libgcj.jar) contains a special zero-length attribute `gnu.gcj.gcj-compiled`. The compiler looks for this attribute when loading `java.lang.Object` and will report an error if this attribute isn't found unless you are compiling Java code to bytecode. The `fforce-classes-archive-check` option can be used to force this checking even when compiling to bytecode.

Creating and Using Jar Files and Shared Libraries

Jar files are compressed archive files in zip format that contain one or more class files and can also contain related data, such as images and sound files that are referenced by the classes contained in the jar file. Jar files simplify distributing Java applications as a single, compressed file that can be directly unpacked and executed by using your Java interpreter's `-jar` option and identifying the name of the jar file.

Jar files are created using the jar tool that is provided as part of any Java development environment. The jar utility lets the programmer create a jar file, add files to an existing jar file, list the contents of an existing jar file, or extract individual files from a jar archive. With one significant exception, the options used for any of the operations are identical to the options used by the standard Unix/Linux tar application. For example, the following commands create a class file for the Fibonacci.java example used throughout this chapter and then create a jar file called fib.jar that contains this class file:

```
$ gcj -C Fibonacci.java
$ jar cvf fib.jar Fibonacci.class
```

```
added manifest
adding: Fibonacci.class(in = 1033) (out= 639)(deflated 38%)
```

However, attempting to execute this jar file produces an error message, as shown in the following example:

```
$ java -jar fib.jar
```

```
Failed to load Main-Class manifest attribute from
fib.jar
```

This message is the jar file's equivalent of the "Exception in thread "main" java.lang.NoClassDefFoundError:" error message that you see when the Java interpreter does not know which class contains the main method that it should invoke as discussed in the sidebar titled "Error Messages for Main Methods," earlier in this chapter. To resolve this problem, you must create a Manifest.txt file that identifies the class containing the `main` method and add this to your jar file. A Manifest.txt file contains a single entry in the following format:

```
Main-Class: package-name.class-name
```

For our simple example, the corresponding Manifest.txt file would look like the following:

```
Main-Class: Fibonacci
```

A manifest.txt file must end with a carriage return/new line combination. To add a Manifest.txt file to a jar file, you identify it using the jar command's `--m` option. To specify a Manifest.txt file when creating a jar file, you would use a command such as the following:

```
$ jar -cvfm fib.jar Manifest.txt Fibonacci.class
added manifest
adding: Fibonacci.class(in = 1033) (out= 639)(deflated 38%)
```

When creating a jar file containing a Manifest.txt file, you must make sure that you specify filenames in the right order. The jar command's `--f` option indicates that the next filename is the name of the archive file. Similarly, the `--m` option indicates that the next available filename identifies the Manifest.txt file. If you specify the `--m` option before the `--f` option, the jar command will misinterpret your command-line arguments and think that the name of your jar file is the Manifest.txt file—since it does not yet exist, you will see an error message such as the following:

```
$ jar -cmvf fib.jar Fibonacci.class Manifest.txt
```

```
java.io.FileNotFoundException: fib.jar (No such file or directory)
        at java.io.FileInputStream.open(Native Method)
        at java.io.FileInputStream.<init>(FileInputStream.java:106)
        at java.io.FileInputStream.<init>(FileInputStream.java:66)
        at sun.tools.jar.Main.run(Main.java:123)
        at sun.tools.jar.Main.main(Main.java:904)
```

Specifying your class file(s) and your Manifest.txt file in the wrong order will produce a similar error message, as in the following example:

```
$ jar -cvfm fib.jar Fibonacci.class Manifest.txt
```

```
java.io.IOException: invalid header field
        at java.util.jar.Attributes.read(Attributes.java:383)
        at java.util.jar.Manifest.read(Manifest.java:167)
        at java.util.jar.Manifest.<init>(Manifest.java:52)
        at sun.tools.jar.Main.run(Main.java:124)
        at sun.tools.jar.Main.main(Main.java:904)
```

To add the Mainfest.txt file to an existing jar file, you would use a command such as the following:

```
$ jar -ufm fib.jar Manifest.txt
```

If you see any error messages, check your Manifest.txt file and make absolutely sure that you have specified the arguments in the right order.

Versions 4.0 and greater of gcj provide a runtime tool to enable natively compiled Java code to be used as a shared library if that code has been compiled with the new `-findirect-dispatch` option. This resolves problems encountered due to the fact that many Java applications have no provisions for handling natively compiled code in the class loader. To create a shared library from a jar file, you would use a command such as the following:

```
$ gcj -shared -findirect-dispatch -fjni -fPIC fib.jar -o fib.jar.so
```

The `-shared` and `-fPIC` options are standard options used when creating shared libraries. The `-findirect-dispatch` option enables the new binary compatibility ABI (application binary interface) used in gcj 4.0 and better, and the `-fjni` option tells gcj to use JNI native methods rather than CNI.

■ **Tip** When creating shared libraries from jar files on ELF platforms, linker options such as `-Wl,-Bsymbolic` (which tell the linker to bind references to global symbols to the definition within the shared library) typically increase performance.

Once you have created the shared library file, you must use the `gcj-dbtool` command to add it to the database that maps jar files to shared libraries. To determine the default database used by gcj on your system, you use the `gcj-dbtool` command's `-p` option, as in the following example:

```
$ gcj-dbtool -p
```

```
/usr/lib64/gcj-4.2/classmap.db
```

You must be root to add entries to the global database, which you do using a command such as the following:

```
$ gcj-dbtool -a /usr/lib64/gcj-4.2/classmap.db fib.jar fib.jar.so
```

You can then access this jar file as a shared library from anywhere on your system. If it contains a main routine, you can also execute it directly.

■ **Tip** The gcj-dbtool application uses the full pathname for the tar shared library entry. Make sure that any shared libraries that you have created from your jar files are built in the location from which you want others to be able to permanently access them.

GCC Java Support and Extensions

As mentioned previously, this book is not a tutorial or reference on writing Java applications—there are already plenty of books that do that job and do it well. However, when writing Java code that you will compile using the GCC Java compiler, you can take advantage of a number of extensions to Java

through both the compiler itself and the standard Java library used by gcj and libgcj. This section highlights the most significant extensions that are provided by both as of the time this book was written and discusses some differences in behavior that you may see between Java as defined by the Java specification and the compilation and behavior of Java code compiled using the GCC Java compiler.

Java Language Standard ABI Conformance

An ABI is the binary flipside of the application programming interface (API) defined by the Java datatypes, classes, and methods in the include files and libraries that are provided by a Java library implementation. A consistent binary interface is required in order for compiled Java applications to conform to the binary conventions used in associated Java libraries and related object files for things such as physical organization, parameter passing conventions, and naming conventions. This consistency is especially critical for specific language support routines, most notably those used for throwing or catching exceptions.

Most recent Java compilers conform to a specific ABI, which effectively determines the execution environments in which code compiled with that compiler can execute correctly (modulo coding errors, of course—perhaps "execute as written" would be a more precise statement). Beginning with version 4.0 of the GCC, gcj conforms to the Java ABI as defined in the Java Language Specification.

Runtime Customization

In addition to the standard Java `CLASSPATH` environment variable, gcj also uses another environment variable, `GCJ_PROPERTIES`, which is consulted at runtime by any binaries generated using gcj. This environment variable is intended to hold a list of assignments to global Java properties, such as those that would be set using a JVM's `-D` option. The following code example provides a simple example of using this environment variable:

```
public class Test {
 public static void main(String[] args) {
   System.out.println("Property foo.bar value is:" + System.getProperty("foo.bar"));
 }
}
```

Compiling this code in the standard manner and executing it with no value for `GCJ_PROPERTIES` does the right thing:

```
$ gcj --main=Test -o wvhTest Test.java
$ ./wvhTest
```

```
Property foo.bar value is: null
```

```
$ export GCJ_PROPERTIES="foo.bar=42"
$ ./wvhTest
```

```
Property foo.bar value is: 42
```

The `GCJ_PROPERTIES` environment variable provides a convenient way to set application-specific property values when doing iterative testing of your Java applications and can also be used to set system properties such as `java.version` when testing existing applications that behave differently depending upon the value of a specific system property.

Getting Information About Java Source and Bytecode Files

GCC distributions that include gcj also include several other useful binaries. Probably the most commonly used of these is the GNU interpreter for Java, gij, which is discussed later in this chapter in the section "Using the GNU Interpreter for Java." But GCC also includes two binaries that you can use to get information about Java source and class files. The next two sections provide general information about using these utilities.

Getting Information About Java Files Using jv-scan

The jv-scan command provides a number of options that display selected information about a Java source file. Table 4-5 summarizes the available options for the jv-scan application.

Table 4-5. *jv-scan Command-Line Options*

Option	Description
`--complexity`	Prints the cyclomatic complexity of the input file, which corresponds to the number of paths through the methods in the input file.
`--encoding` *NAME*	Specifies the encoding used in the input file.
`--help`	Prints a help message that summarizes jv-scan options and then exits.
`--list-class`	Lists all of the classes defined in the input file.
`--list-filename`	Includes the name of the input file when listing class names using the `--list-class` option.
`--no-assert`	Doesn't recognize the `assert` keyword.
`-o` *FILE*	Specifies that jv-scan output be directed to the file *FILE* rather than to stdout.
`--print-main`	Prints the name of the class containing a `main` method.
`--version`	Prints the jv-scan version number and then exits.

Though much of this information can easily be determined by visually scanning your input file, the jv-scan utility is useful with complex input files containing multiple subclasses, especially for simplifying automated compilation, such as make and Ant compilation rules, as suggested by the following two equivalent statements:

```
$ gcj Fibonacci.java --main=Fibonacci-o Fibonacci
$ gcj Fibonacci.java --main=`jv-scan --print-main Fibonacci.java` -o Fibonacci
```

Getting Information About Class Files Using jcf-dump

Conceptually similar to the jv-scan command, the jcf-dump application displays information about an existing class file containing platform-independent Java bytecode. The jcf-dump command provides a number of options that display selected information about a Java class file. Table 4-6 summarizes the available options for the jcf-dump application.

Table 4-6. *jcf-dump Command-Line Options*

Option	Description
`--bootclasspath` *PATH*	Overrides the built-in class path, as discussed earlier in this chapter in the section titled "Constructing the Java Classpath."
`-c`	Includes disassembled method bodies in the jcf-dump output.
`--classpath` *PATH*	Sets path to find `.class` files, as discussed earlier in this chapter in the section titled "Constructing the Java Classpath."
`--extdirs` *PATH*	Sets the path to the extensions directory, as discussed earlier in this chapter in the section titled "Constructing the Java Classpath."
`--help`	Prints a help message that summarizes jcf-dump options and then exits.
`-I`*DIR*	Appends directory to the classpath, as discussed earlier in this chapter in the section titled "Constructing the Java Classpath" and more generally in Appendix A.
`--javap`	Generates output in the format used by the standard javap Java class file disassembler.
`-o` *FILE*	Specifies that jcf-dump output be directed to the file *FILE* rather than to stdout.
`-v`, `--verbose`	Print extra information while running.
`--version`	Prints the jcf-dump version number and then exits.

The jcf-dump command can be very useful when you are working with precompiled class files, or when you simply need summary information about the class files and methods defined in accompanying Java source files. The following is sample output from the jcf-dump application when examining the Fibonacci.class file used earlier in this chapter:

```
$ jcf-dump Fibonacci.class
Reading .class from Fibonacci.class.
Magic number: 0xcafebabe, minor_version: 3, major_version: 45.

Access flags: 0x21 public super
This class: Fibonacci, super: java.lang.Object
Interfaces (count: 0):

Fields (count: 0):

Methods (count: 3):

Method name:"calcFibonacci" Signature: (int)int
Attribute "Code", length:55, max_stack:5, max_locals:2, code_length:23
Attribute "LineNumberTable", length:14, count: 3

Method name:"<init>" public Signature: (int)void
Attribute "Code", length:91, max_stack:4, max_locals:3, code_length:55
Attribute "LineNumberTable", length:18, count: 4
```

```
Method name:"main" public static Signature: (java.lang.String[])void
Attribute "Code", length:76, max_stack:5, max_locals:1, code_length:40
Attribute "LineNumberTable", length:18, count: 4

Attributes (count: 1):
Attribute "SourceFile", length:2, #67="Fibonacci.java"
```

Using the GNU Interpreter for Java

The GNU interpreter for Java, gij, is a binary Java interpreter that is included with GCC distributions that include Java support. The gij binary is a drop-in replacement for existing JVMs and other Java interpreters, modulo differences between the standard Java API and that provided by the Classpath project used by all GCC Java tools. In many cases, gij makes it easy for you to test existing class files in the gcj environment. For example, popular Java applications such as Eclipse are often executed using gij rather than a standard Java VM in order to deliver easily shippable, completely open source solutions.

The gij interpreter provides a number of command-line options that are summarized in Table 4-7. In general, using gij from the command line to execute an existing class or jar file works exactly like using a standard JVM, as shown in the following example output:

```
$ javac hello.java
$ java Hello
```

```
Hello World!
```

```
$ gij Hello
```

```
Hello World!
```

The gcj compiler's `-lgij` command causes output binaries to link with the gij library, which gives those binaries the same execution characteristics used when executing class files with any JVM or Java interpreter, as shown in the following example:

```
$ gcj -lgij hello.java -o gij_hello
$./gij_hello Hello
```

```
Hello World!
```

Table 4-7. *gij Command-Line Options*

Option	Description
`-agentlib`	This option is accepted but ignored by gij, and is present only for compatibility with existing application launch scripts that you want to test with gij rather than a JVM or other Java interpreter.
`-agentpath`	This option is accepted but ignored by gij, and is present only for compatibility with existing application launch scripts that you want to test with gij rather than a JVM or other Java interpreter.

Table 4-7. *gij Command-Line Options (Continued)*

Option	Description
`-client`	This option is accepted but ignored by gij, and is present only for compatibility with existing application launch scripts that you want to test with gij rather than a JVM or other Java interpreter.
`-cp` *PATH*, `-classpath` *PATH*	These options set the initial classpath used for finding class and resource files. If specified, this option overrides any value set using the `CLASSPATH` environment variable. This option is ignored if the `-jar` option is specified. Note that gij does not support any of the other options used by gcj to modify the directories and search order in the Java classpath, as discussed earlier in this chapter in the section titled "Constructing the Java Classpath."
`-d32`	This option is accepted but ignored by gij, and is present only for compatibility with existing application launch scripts that you want to test with gij rather than a JVM or other Java interpreter.
`-d64`	This option is accepted but ignored by gij, and is present only for compatibility with existing application launch scripts that you want to test with gij rather than a JVM or other Java interpreter.
`-D`*name*`[=`*value*`]`	This option defines a global system property named *name* with the value *value*. If no value is specified, the default value of the specified property is the empty string. System properties are initialized at the program's startup and can be retrieved at runtime using the `java.lang.System.getProperty` method.
`-debug`	This option is accepted but ignored by gij, and is present only for compatibility with existing application launch scripts that you want to test with gij rather than a JVM or other Java interpreter.
`--fullversion`	This option causes gij to display detailed version information and then exit.
`--help`, `-?` .	These options cause gij to display a short usage message and then exit.
`-hotspot`	This option is accepted but ignored by gij, and is present only for compatibility with existing application launch scripts that you want to test with gij rather than a JVM or other Java interpreter.
`-jar`	This option tells gij that the filename specified on the command line is the name of a jar file, not a class file.
`-javaagent`	This option is accepted but ignored by gij, and is present only for compatibility with existing application launch scripts that you want to test with gij rather than a JVM or other Java interpreter.
`-jrockit`	This option is accepted but ignored by gij, and is present only for compatibility with existing application launch scripts that you want to test with gij rather than a JVM or other Java interpreter.
`-ms=`*number*	This option is equivalent to `-Xms`*number* and sets the initial heap size allocated by gij to *size*.
`-mx=`*number*	This option is equivalent to `-Xmx`*number* and sets the maximum size of the heap that can be allocated by gij to *size*.

Table 4-7. *gij Command-Line Options (Continued)*

Option	Description
`-noclassgc`	This option is accepted but ignored by gij, and is present only for compatibility with existing application launch scripts that you want to test with gij rather than a JVM or other Java interpreter.
`-noverify`	This option does not verify types or the compliance of Java bytecode with the Java Virtual Machine specification.
`-server`	This option is accepted but ignored by gij, and is present only for compatibility with existing application launch scripts that you want to test with gij rather than a JVM or other Java interpreter.
`--showversion`	This option prints the gij version number and then continues execution. This option is useful for capturing version number information in scripted test output.
`-verbose`, `-verbose:class`	These options display a short message on stderr as each class is initialized.
`-verify`	This option is accepted but ignored by gij, and is present only for compatibility with existing application launch scripts that you want to test with gij rather than a JVM or other Java interpreter.
`-verifyremote`	This option is accepted but ignored by gij, and is present only for compatibility with existing application launch scripts that you want to test with gij rather than a JVM or other Java interpreter.
`--version`	This option displays the gij version number and then exits.
`-X`	This option causes gij to list all the supported options for setting execution parameters.
`-Xms`*size*	This option sets the initial heap size allocated by gij to *size*.
`-Xmx`*size*	This option sets the maximum size of the heap that can be allocated by gij to *size*.

Java and C++ Integration Notes

In addition to the standard JNI defined by the Java specification as a standard for writing native Java methods in C or C++, gcj provides an alternative Compiled Native Interface (CNI), which originally stood for the Cygnus Native Interface. This interface takes advantage of the fact that much of GCC's Java support is very similar to its support for C++. This interface also makes it easy to call classes and methods written in C++, which is especially convenient if you already have access to C++ code that does what you want to do.

Because GCC's C++ and Java support use the same calling conventions and data layout, interoperability between C++ and Java requires little manual intervention. CNI provides C++ classes for primitive Java types, includes functions to work with Java strings and arrays, delivers the `gcjh` command to generate C++ header files to map Java reference types to C++ types, and provides an API for calling Java methods from C++ programs. Extensive and detailed CNI documentation is provides online as part of the gcj sections of the latest GCC documentation at `http://gcc.gnu.org/onlinedocs`.

One significant consideration is that the gcj runtime requires a certain amount of initialization, which you can make sure will happen if your `main` method is a Java method. If you prefer to call Java from C++ `main` code, you will have to use functions such as `JvCreateJavaVM()` and `JvAttachCurrentThread()` to manually create a virtual JVM instance and attach to it (which will do the necessary initialization) before making Java calls. See the extensive (and excellent) CNI documentation for additional details.

CHAPTER 5

Optimizing Code with GCC

These days, compilers are pretty smart. They can perform all sorts of code transformations—from simple inlining to sophisticated register analysis—that make compiled code run faster. In most situations, faster is better than smaller, because disk space and memory are quite cheap for desktop users. However, for embedded systems small is often at least as important as fast because of a commonplace environment consisting of extreme memory constraints and no disk space, making code optimization a very important task.

By this point in the book, you have a pretty good grasp of how to compile your code and how to make gcc, the GCC C compiler, do as you please. The next step, accordingly, is to make your code faster or smaller, which is the topic of this chapter. Based on what you learn, you might even be able to make your next program faster *and* smaller. After a quick, high-level overview of compiler optimization theory, we'll discuss GCC's command-line options for code optimization, starting with general, architecture-independent optimizations and concluding with architecture-specific optimizations.

While the examples in this chapter are given in the C programming language, the optimization options discussed in this chapter are independent of the programming language in which your code is written. Being able to share optimization flags across compilers for different languages is a significant advantage of using a suite of compilers, such as those provided by GCC (GNU Compiler Collection).

OPTIMIZATION AND DEBUGGING

In the absence of optimization, GCC's goal, besides compiling code that works, is to keep compilation time to a minimum and generate code that runs predictably in a debugging environment. For example, in optimized code, a variable whose value is repeatedly computed inside a loop might be moved above the loop if the optimizer determines that its value does not change during the loop sequence. Although this is acceptable (if, of course, it does not alter the program's results), this optimization makes it impossible to debug the loop as expressed in your source code because you cannot set a breakpoint on that variable to halt execution of the loop. Without optimization, on the other hand, you can set a breakpoint on the statement computing the value of this variable, examine variables, and then continue execution. This is what is meant by "code that runs predictably in a debugging environment."

As is discussed in this chapter, optimization can change the flow, but not the results, of your code. For this reason, optimization is a phase of development generally best left until after you have completely written and debugged your application. Your mileage may vary, but it can be tricky to debug code that has been optimized by any of the GCC compilers.

A Whirlwind Tour of Compiler Optimization Theory

Optimization refers to analyzing a straightforward compilation's resulting code to determine how to transform it to run faster, consume fewer resources, or both. Compilers that do this are known as *optimizing compilers,* and the resulting code is known as *optimized code*. To produce optimized code, an optimizing compiler performs one or more transformations on the source code provided as input. The purpose is to replace less efficient code with more efficient code while at the same time preserving the meaning and ultimate results of the code. Throughout this section, I will use the term *transformation* to refer to the code modifications performed by an optimizing compiler because I want to distinguish between optimization techniques, such as loop unrolling and common subexpression elimination, and the way in which these techniques are implemented using specific code transformations.

Optimizing compilers use several methods to determine where generated code can be improved. One method is *control flow analysis*, which examines loops and other control constructs, such as `if-then` and `case` statements, to identify the execution paths a program might take and, based on this analysis, determine how the execution path can be streamlined. Another typical optimization technique examines how data is used in a program, a procedure known as *data flow analysis*. Data flow analysis examines how and when variables are used (or not) in a program and then applies various set equations to these usage patterns to derive optimization opportunities.

Optimization properly includes improving the algorithms that make up a program as well as the mechanical transformations described in this chapter. The classic illustration of this is the performance of bubble sorts compared to, say, quick sorts or shell sorts. While code transformations might produce marginally better runtimes for a program that sorts 10,000 items using a bubble sort, replacing the naive bubble sort algorithm with a more sophisticated sorting algorithm will produce dramatically better results. The point is simply that optimization requires expending both CPU cycles and human intellectual energy.

For the purposes of this section, a *basic block* is a sequence of consecutive statements entered and exited without stopping or branching elsewhere in the program. Transformations that occur in the context of a basic block, that is, within a basic block, are known as *local transformations*. Similarly, transformations that do not occur solely within a basic block are known as *global transformations*. As it happens, many transformations can be performed both locally and globally, but the usual procedure is to perform local transformations first.

Although the examples in this section use C, all GCC compilers actually perform transformations on intermediate representations of your program. These intermediate representations are better suited for compiler manipulations than unmodified language-specific source code. GCC uses a sequence of different intermediate representations of your code before generating binaries, as discussed later in this chapter in "GCC Optimization Basics." I use C source code in the examples in this chapter rather than any intermediate representation because, well, people write code in C and other high-level languages, and not directly in any intermediate representation.

GCC compilers perform many other types of optimization, some extremely granular and best left to dedicated discussions of compiler theory. The optimizations listed in this section are some of the more common optimizations that GCC compilers can perform based on the command-line switches that are discussed later in this chapter.

■ **Note** The classic compiler reference, affectionately known as "the dragon book" because of the dragon on its cover, is *Compilers: Principles, Techniques, and Tools*, Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman (Addison Wesley Longman, 1986. ISBN: 0-201-10088-6). This seminal text goes into much more detail than this chapter on various optimization techniques; and I'm indebted to it for teaching me this stuff in the first place.

Code Motion

Code motion is an optimization technique that is related to eliminating redundant code through common subexpression elimination (discussed later in this chapter). Code motion does not entirely remove common subexpressions, but attempts to reduce the number of times they occur by relocating them to more optimal locations in an intermediate representation of the code. For example, in nested loops or other control constructs, intermediate value calculations may be performed more times than necessary. In order to optimize such programs, compilers can move these calculations to locations where they will be executed less frequently, but will still produce the same result in the compiled code. Code motion that specifically moves redundant calculations outside the scope of a loop is referred to as *loop-invariant code motion*. Code motion is also used in an optimization technique that is related to common subexpression elimination known as *partial redundancy elimination*.

Common Subexpression Elimination

Eliminating redundant computations is a standard optimization mechanism because it reduces the number of instructions that a program has to execute in order to arrive at the same result. For example, given an expression whose value is computed only once, each subsequent occurrence of that expression can be eliminated and its value used instead, if the values of the variables in the expression have not changed since first computed. The subsequent occurrences of the expression are called *common subexpressions*. Consider the code fragment in Listing 5-1.

Listing 5-1. *An Example of a Common Subexpression*

```
#define STEP 3
#define SIZE 100

int i, j, k;
int p[SIZE];

for (i = 0; i < SIZE; ++i) {
        j = 2 * STEP;
        k = p[i] * j;
}
```

The expression `j = 2 * STEP` in the `for` loop is a common subexpression because its value is computed before entering the loop and its constituent variable, `STEP` (actually a predefined value), has not changed. Common subexpression elimination (CSE) removes the repeated computation of `j` in the `for` loop. After CSE, the `for` loop might look like the following:

```
j = 2 * STEP;
for (i = 0; i < 100; ++i) {
        k = p[i] * j;
}
```

Admittedly, the example is contrived, but it makes the point: CSE has eliminated 100 redundant computations of `j`. CSE improves performance by eliminating unnecessary computations and typically also reduces the size of the resulting binary by eliminating unnecessary instructions.

Constant Folding

Constant folding is an optimization technique that eliminates expressions that calculate a value that can already be determined when the program is compiled. These are typically calculations that only

reference constant values or expressions that reference variables whose values are constant. For example, both of the calculations in the following code fragment could be replaced with a simple assignment statement:

```
n = 10 * 20 * 400;

i = 10;
j = 20;
ij = i * j;
```

In the latter case, the assignments to the variables `i` and `j` could also be eliminated if these variables were not used elsewhere in the code.

Copy Propagation Transformations

Another way to reduce or eliminate redundant computations is to perform copy propagation transformations, which eliminate cases in which values are copied from one location or variable to another in order to simply assign their value to another variable. Given an assignment of the form `f = g`, subsequent uses of `f`, called *copies*, use `g` instead. Consider a code fragment such as the following:

```
i = 10;
x[j] = i;
y[MIN] = b;
b = i;
```

Copy propagation transformation might result in a fragment that looks like the following:

```
i = 10;
x[j] = 10;
y[MIN] = b;
b = 10;
```

In this example, copy propagation enables the code to assign values from a static location, which is faster than looking up and copying the value of a variable, and also saves time by eliminating assigning a value to a variable that is itself subsequently used only to propagate that value throughout the code. In some cases, copy propagation itself may not provide direct optimizations, but simply facilitates other transformations, such as code motion and dead code elimination.

Constant propagation is a related optimization to copy propagation transformation that optimizes code by replacing variables with constant values wherever possible. Copy propagation focuses on eliminating needless copies between different variables, while constant propagation focuses on eliminating needless copies between predefined values and variables.

Dead Code Elimination

Dead code elimination (DCE) is the term used for optimizations that remove code that doesn't actually do anything permanent or useful. You might wonder why you'd write this type of source code, but it can easily creep into large, long-lived programs even at the source code level. Much dead code elimination is actually done on intermediate representations of your code, which may contain unnecessary intermediate calculations, simply because they are more generic representations of your source code.

Unreachable code elimination is a related optimization that removes code that the compiler can identify as impossible to reach. For example, consider the following sample block:

```
if ( i == 10 ) {
    . . .
} else {
    . . .
    if ( i == 10) {
       . . .
    }
}
```

In this example, the nested test for whether `i` is equal to 10 can be eliminated because it can never be reached, since that case would be caught by the first clause of the enclosing `if` construct. Unreachable code elimination is much more likely to occur in the intermediate forms of your code that are generated as part of the compilation process than on anything that is directly visible in your source code.

If-Conversion

If-conversion is a technique where branch constructs, such as large if-then-elseif-else constructs, are broken into separate `if` statements to simplify generated code, provide opportunities for further optimization, and eliminate jumps and branches wherever possible.

Inlining

Inlining is an optimization technique where code performance can improve by replacing complex constructs and even function calls with inline representations of the construct or function call. *Code inlining*, or *loop unrolling*, are terms for replacing all or portions of a loop with an explicit series of instructions. *Function inlining* is the term for replacing function calls with the explicit set of instructions that are performed by the function. In general, inlining can reduce code complexity and provide higher performance than would be required by branching that would be done otherwise. It often also provides opportunities for common subexpression elimination and code motion optimization. The classic example of optimization through inlining and unrolling is Duff's Device, which is explained at `http://en.wikipedia.org/wiki/Duff's_device` in more detail than would be useful here.

GCC Optimization Basics

GCC uses a sequence of different intermediate representations of your code before generating binaries. Converting language-specific code into simpler forms provides several primary advantages:

- Breaking your code into simpler, more low-level constructs exposes opportunities for optimization that may not be directly visible in your source code.
- Using intermediate representations enables the simpler form to more easily and readably represent parse trees that are otherwise at varying levels of complexity.
- Using a simpler form enables the GCC compilers to share optimization mechanisms, regardless of the language in which your code was originally written.

Traditionally, GCC has always internally used an intermediate form known as *Register Transfer Language* (RTL), which is a very low-level language into which all code compiled with a GCC compiler (in any high-level language) is transformed before object code generation begins. Transforming high-level input languages into an intermediate, internal form is a time-tested mechanism for exposing opportunities for optimization. As a very low-level representation of your code, GCC's RTL lends itself well to optimizations that are similarly low-level, such as register allocation and stack and data

optimizations. Because it is relatively low-level, RTL is not as good as one would like in terms of supporting higher-level optimizations related to data types, array and variable references, and overall data and control flow within your code.

With GCC 4.0, the fine makers of the GCC compilers introduced a new intermediate form, static single assignment (SSA), which operates on the parse trees that are produced from your code by GCC compilers, and is thus known as Tree SSA. Without clubbing you to death with algorithms, it is interesting to note that 4.0 and later GCC compilers use two intermediate forms before arriving at a Tree SSA representation, known as GENERIC and GIMPLE. A GENERIC representation is produced by eliminating language-specific constructs from the parse tree that is generated from your code, and a GIMPLE representation is then produced from the GENERIC representation by simplifying address references within the code. Even semantically, you can see that introducing these two intermediate forms and the Tree SSA representation provides several additional points at which optimizations can occur at as high a level as possible before zooming into the depths of the RTL representation of your code.

The best source of information on Tree SSA and the actual steps of the optimization process can be found in various papers and presentations by its authors. One particularly interesting presentation was made at the 2003 GCC Developers' Summit, and is part of the proceedings for that conference. You can find these online at `http://www.linux.org.uk/~ajh/gcc/gccsummit-2003-proceedings.pdf`.

What's New in GCC 4.x Optimization

The most significant changes to GCC optimization in the GCC 4.x family are related to the introduction of the intermediate Tree SSA format discussed in the previous section. This has provided many new opportunities for optimization and therefore introduced many new options, including `-ftree-ccp`, `-ftree-ch`, `-ftree-copyrename`, `-ftree-dce`, `-ftree-dominator-opts`, `-ftree-dse`, `-ftree-fre`, `-ftree-loop-im`, `-ftree-loop-ivcanon`, `-ftree-loop-linear`, `-ftree-loop-optimize`, `-ftree-lrs`, `-ftree-pre`, `-ftree-sra`, `-ftree-ter`, and `-ftree-vectorize`, which are explained later in this chapter. Due largely to the introduction of these new options, the default optimizations that are invoked by the generic `-O1`, `-O2`, `-O3`, and `-Os` optimization levels have also changed. Another side effect of introducing the intermediate Tree SSA form is that optimization is more consistent, regardless of the input language and GCC compiler you are using.

GCC 4 also introduces substantial improvements in vectorization, thanks largely to contributions from IBM. Vectorization increases the efficiency of processor operations by finding code where a single instruction can be applied to multiple data elements. GCC 4 enables up to 16 scalar operations to be mapped to a single vector operation. This optimization can be especially useful in gaming, graphics, and multimedia applications where instructions are repetitively applied to arrays of values.

GCC 4 also introduces improvements in checking array boundaries and validating the contents and structure of the stack, both of which help automate protections against popular application attacks that attempt to induce buffer or stack overflows.

Architecture-Independent Optimizations

GCC's optimization knobs and switches fall into two broad categories: architecture-independent and architecture-specific optimizations. This section covers the architecture-independent optimizations. These optimizations do not depend on features specific to a given architecture, such as x86; class of processors, such as Intel IA-32 CPUs; and characteristics of a given instance of a processor family, such as a Pentium IV (Xeon).

GCC's most well-known optimization switches are `-O`; its variant `-On`, where `n` is an integer between 0 and 3; and `-Os`. `-O0` turns off optimization. `-O` and `-O1` (which I will call *level 1 optimizations*) are equivalent, telling GCC to optimize code. With `-O` or `-O1`, the compiler attempts to minimize both

code size and execution time without dramatically increasing compilation time. Using `-O2` and `-O3` will increase the optimization level from that requested by `-O1`, while still invoking the optimization options requested by `-O1`. To minimize code size, use option `-Os`.

The tables in this section show the optimization options associated with various GCC optimization levels. To disable one of them while leaving the others enabled, negate the option using `no-` between `-f` and the optimization name. For example, to disable deferred stack pops, the command line might resemble this:

```
$ gcc myprog.c -o myprog -O1 -fno-defer-pop
```

Note `-f` denotes a flag whose operation is machine independent, that is, it requests an optimization that can be applied regardless of the architecture (in most cases). Flags, or flag options, modify GCC's default behavior at a given optimization level but do not require specific hardware support to implement the optimization. As usual, you can specify multiple flags as needed.

Level 1 GCC Optimizations

The optimizations listed in Table 5-1 are enabled by default when you specify the `-O` or `-O1` optimization options.

Table 5-1. *Optimizations Enabled with -O and -O1*

Optimization	Description
`-fcprop-registers`	Attempts to reduce the number of register copy operations performed.
`-fdefer-pop`	Accumulates function arguments on the stack.
`-fdelayed-branch`	Utilizes instruction slots available after delayed branch instructions.
`-fguess-branch-probability`	Uses a randomized predictor to guess branch possibilities.
`-fif-conversion`	Converts conditional jumps into nonbranching code.
`-fif-conversion2`	Performs if-conversion using conditional execution (on CPUs that support it).
`-floop-optimize`	Applies several loop-specific optimizations.
`-fmerge-constants`	Merges identical constants used in multiple modules.
`-fomit-frame-pointer`	Omits storing function frame pointers in a register. Only activated on systems where this does not interfere with debugging.
`-ftree-ccp`	Performs sparse conditional constant propagation (CCP) on SSA trees (GCC 4.x only).
`-ftree-ch`	Performs loop header copying on SSA trees, which eliminates a jump and provides opportunities for subsequent code motion optimization (GCC 4.x only).
`-ftree-copyrename`	Performs copy renaming on SSA trees, which attempts to rename internal compiler temporary names at copy location to names that more closely resemble the original variable names (GCC 4.x only).

Table 5-1. *Optimizations Enabled with -O and -O1 (Continued)*

Optimization	Description
`-ftree-dce`	Performs dead code elimination (DCE) on SSA trees (GCC 4.x only).
`-ftree-dominator-opts`	Performs a variety of optimizations using a dominator tree traversal. A dominator tree is a tree where each node's children are the nodes that it immediately dominates. These cleanups include constant/copy propagation, redundancy elimination, range propagation, expression simplification, and jump threading (reducing jumps to other jumps) (GCC 4.x only).
`-ftree-dse`	Performs dead store elimination (DSE) on SSA trees (GCC 4.x only).
`-ftree-fre`	Performs full redundancy elimination (FRE) on SSA trees, which only considers expressions that are computed on full paths leading to the redundant compilation. This is similar to and faster than a full partial redundancy elimination (PRE) pass, but discovers fewer redundancies than PRE (GCC 4.x only).
`-ftree-lrs`	Performs live range splitting when converting SSA trees back to normal form prior to RTL generation. This creates unique variables in distinct live ranges where a variable is used, providing subsequent opportunities for optimization (GCC 4.x only).
`-ftree-sra`	Performs scalar replacement of aggregates, which replaces structure references with scalar values to avoid committing structures to memory earlier than necessary (GCC 4.x only).
`-ftree-ter`	Performs temporary expression replacement (TER) when converting SSA trees back to normal form prior to RTL generation. This replaces single-use temporary expressions with the expressions that defined them, making it easier to generate RTL code and provide opportunities for better subsequent optimization in the RTL code (GCC 4.x only).

Level 1 optimizations comprise a reasonable set of optimizations that include both size reduction and speed enhancement. For example, `-tree-dce` eliminates dead code in applications compiled with GCC 4, thereby reducing the overall code size. Fewer jump instructions mean that a program's overall stack consumption is smaller. `-fcprop-registers`, on the other hand, is a performance optimization that works by minimizing the number of times register values are copied around, saving the overhead associated with register copies.

`-fdelayed-branch` and `-fguess-branch-probability` are instruction scheduler enhancements. If the underlying CPU supports instruction scheduling, these optimization flags attempt to utilize the instruction scheduler to minimize CPU delays while the CPU waits for the next instruction.

The loop optimizations applied when you specify `-floop-optimize` include moving constant expressions above loops and simplifying test conditions for exiting loops. At level 2 and higher optimization levels, this flag also performs strength reduction and unrolls loops.

`-fomit-frame-pointer` is a valuable and popular optimization for two reasons: it avoids the instructions required to set up, save, and restore frame pointers and, in some cases, makes an additional CPU register available for other uses. On the downside, in the absence of frame pointers, debugging (such as generating stack traces, especially from deeply nested functions) can be difficult if not impossible.

-O2 optimization (level 2 optimization) includes all level 1 optimizations plus the additional optimizations listed in Table 5-2. Applying these optimizations will lengthen the compile time, but as a result you should also see a measurable increase in the resulting code's performance, or, rather, a measurable decrease in execution time.

Level 2 GCC Optimizations

The optimizations listed in Table 5-2 are enabled by default when you specify the -O2 optimization option.

Table 5-2. *Optimizations Enabled with -O2*

Optimization	Description
-falign-functions	Aligns functions on powers-of-2 byte boundaries.
-falign-jumps	Aligns jumps on powers-of-2 byte boundaries.
-falign-labels	Aligns labels on powers-of-2 byte boundaries.
-falign-loops	Aligns loops on powers-of-2 byte boundaries.
-fcaller-saves	Saves and restores register values overwritten by function calls.
-fcrossjumping	Collapses equivalent code to reduce code size.
-fcse-follow-jumps	Follows jumps whose targets are not otherwise reached.
-fcse-skip-blocks	Follows jumps that conditionally skip code blocks.
-fdelete-null-pointer-checks	Eliminates unnecessary checks for null pointers.
-fexpensive-optimizations	Performs "relatively expensive" optimizations.
-fforce-mem	Stores memory operands in registers (obsolete in GCC 4.1).
-fgcse	Executes a global CSE pass.
-fgcse-lm	Moves loads outside of loops during global CSE.
-fgcse-sm	Moves stores outside of loops during global CSE.
-foptimize-sibling-calls	Optimizes sibling and tail recursive function calls.
-fpeephole2	Performs machine-specific peephole optimizations.
-fregmove	Reassigns register numbers for maximum register tying.
-freorder-blocks	Reorders basic blocks in the compiled function in order to reduce branches and improve code locality.
-freorder-functions	Reorders basic blocks in the compiled function in order to improve code locality by using special text segments for frequently and rarely executed functions.
-frerun-cse-after-loop	Executes the CSE pass after running the loop optimizer.
-frerun-loop-opt	Executes the loop optimizer twice.
-fsched-interblock	Schedules instructions across basic blocks.
-fsched-spec	Schedules speculative execution of nonload instructions.

Table 5-2. *Optimizations Enabled with -O2 (Continued)*

Optimization	Description
`-fschedule-insns`	Reorders instructions to minimize execution stalls.
`-fschedule-insns2`	Performs a second `schedule-insns` pass.
`-fstrength-reduce`	Replaces expensive operations with cheaper instructions.
`-fstrict-aliasing`	Instructs the compiler to assume the strictest possible aliasing rules.
`-fthread-jumps`	Attempts to reorder jumps so they are arranged in execution order.
`-ftree-pre`	Performs partial redundancy elimination (PRE) on SSA Trees.
`-funit-at-a-time`	Parses the entire file being compiled before beginning code generation, enabling extra optimizations such as reordering code and declarations, and removing unreferenced static variables and functions.
`-fweb`	Assigns each web (the live range for a variable) to its own pseudo-register, which can improve subsequent optimizations such as CSE, dead code elimination, and loop optimization.

The four `-falign-` optimizations force functions, jumps, labels, and loops, respectively, to be aligned on boundaries of powers of 2. The rationale is to align data and structures on the machine's natural memory size boundaries, which should make accessing them faster. The assumption is that code so aligned will be executed often enough to make up for the delays caused by the no-op instructions necessary to obtain the desired alignment.

`-fcse-follow-jumps` and `-fcse-skip-blocks`, as their names suggest, are optimizations performed during the CSE optimization pass described in the first section of this chapter. With `-fcse-follow-jumps`, the optimizer follows jump instructions whose targets are otherwise unreached. For example, consider the following conditional:

```
if (i < 10) {
        foo();
} else {
        bar();
}
```

Ordinarily, if the condition (`i < 10`) is false, CSE will follow the code path and jump to `foo()` to perform the CSE pass. If you specify `-fcse-follow-jumps`, though, the optimizer will not jump to `foo()` but to the jump in the `else` clause (`bar()`).

`-fcse-skip-blocks` causes CSE to skip blocks conditionally. Suppose you have an `if` clause with no `else` statement, such as the following:

```
if (i >= 0) {
        j = foo(i);
}
bar(j);
```

If you specify `-fcse-skip-blocks`, and if, in fact, i is negative, CSE will jump to `bar()`, bypassing the interior of the `if` statement. Ordinarily, the optimizer would process the body of the `if` statement, even if `i` tests false.

■ **Tip** If you use computed `goto`s, GCC extensions discussed in Chapter 2, you might want to disable global CSE optimization using the `-fno-gcse` flag. Disabling global CSE in code using computed `goto`s often results in better performance in the resulting binary.

`-fpeephole2` performs CPU-specific peephole optimizations. During peephole optimization, the compiler attempts to replace longer sets of instructions with shorter, more concise instructions. For example, given the following code:

```
a = 2;
for (i = 1; i < 10; ++i)
        a += 2;
```

GCC might replace the loop with the simple assignment `a = 20`. With `-fpeephole2`, GCC performs peephole optimizations using features specific to the target CPU instead of, or in addition to, standard peephole optimization tricks such as, in C, replacing arithmetic operations with bit operations where this improves the resulting code.

`-fforce-mem` copies memory operands and constants into registers before performing pointer arithmetic on them. The idea behind these optimizations is to make memory references common subexpressions, which can be optimized using CSE. As explained in the first section of this chapter, CSE can often eliminate multiple redundant register loads, which incur additional CPU delays due to the load-store operation.

`-foptimize-sibling-calls` attempts to optimize away tail recursive or sibling call functions. A *tail recursive call* is a recursive function call made in the tail of a function. Consider the following code snippet:

```
int inc(int i)
{
    printf("%d\n" i);
    if(i < 10)
        inc(i + 1);
}
```

This defines a function named `inc()` that displays the value of its argument, `i`, and then calls itself with 1 + its argument, `i + 1`, as long as the argument is less than 10. The tail call is the recursive call to `inc()` in the tail of the function. Clearly, though, the recursive sequence can be eliminated and converted to a simple series of iterations because the depth of recursion is fixed. `-foptimize-sibling-calls` attempts to perform this optimization. A *sibling call* refers to function calls made in a tail context that can also be optimized away.

GCC Optimizations for Code Size

The `-Os` option is becoming increasingly popular because it applies all of the level 2 optimizations except those known to increase the code size. `-Os` also applies additional techniques to attempt to reduce the code size. Code size, in this context, refers to a program's memory footprint at runtime rather than its on-disk storage requirement. In particular, `-Os` disables the following optimization flags (meaning they will be ignored if specified in conjunction with `-Os`):

- `-falign-functions`
- `-falign-jumps`
- `-falign-labels`
- `-falign-loops`

- `-fprefetch-loop-arrays`
- `-freorder-blocks`
- `-freorder-blocks-and-partition`
- `-ftree-ch`

You might find it instructive to compile a program with `-O2` and `-Os` and compare runtime performance and memory footprint. For example, I have found that recent versions of the Linux kernel have nearly the same runtime performance when compiled with `-O2` and `-Os`, but the runtime memory footprint is 15 percent smaller when the kernel is compiled with `-Os`. Naturally, your mileage may vary and, as always, if it breaks, you get to keep all of the pieces.

Level 3 GCC Optimizations

Specifying the `-O3` optimization option enables all level 1 and level 2 optimizations plus the following:

- `-fgcse-after-reload`: Performs an extra load elimination pass after reloading
- `-finline-functions`: Integrates all "simple" functions into their callers
- `-funswitch-loops`: Moves branches with loop invariant conditions out of loops

---

■ **Note** If you specify multiple `-O` options, the last one encountered takes precedence. Thus, given the command `gcc -O3 foo.c bar.c -O0 -o baz`, no optimization will be performed because `-O0` overrides the earlier `-O3`.

Manual GCC Optimization Flags

In addition to the optimizations enabled using one of the `-O` options, GCC has a number of specialized optimizations that can only be enabled by specifically requesting them using `-f`. Table 5-3 lists these options.

Table 5-3. *Specific GCC Optimizations*

Flag	Description
`-fbounds-check`	Generates code to validate indices used for array access.
`-fdefault-inline`	Compiles C++ member functions inline by default.
`-ffast-math`	Sets `-fno-math-errno`, `-funsafe-math-optimizations`, and `-fno-trapping-math`.
`-ffinite-math-only`	Disables checks for NaNs and infinite arguments and results.
`-ffloat-store`	Disables storing floating-point values in registers.
`-fforce-addr`	Stores memory constants in registers.
`-ffunction-cse`	Stores function addresses in registers.
`-finline`	Expands functions inline using the `inline` keyword.
`-finline-functions`	Expands simple functions in the calling function.

Table 5-3. *Specific GCC Optimizations*

Flag	Description
`-finline-limit=n`	Limits inlining to functions no greater than `n` pseudo-instructions.
`-fkeep-inline-functions`	Keeps inline functions available as callable functions.
`-fkeep-static-consts`	Preserves unreferenced variables declared `static const`.
`-fmath-errno`	Sets `errno` for math functions executed as a single instruction.
`-fmerge-all-constants`	Merges identical variables used in multiple modules.
`-ftrapping-math`	Emits code that generates user-visible traps for FP operations.
`-ftrapv`	Generates code to trap overflow operations on signed values.
`-funsafe-math-optimizations`	Disables certain error checking and conformance tests on floating-point operations.

Many of the options listed in Table 5-3 involve floating-point operations. In order to apply the optimizations in question, the optimizer deviates from strict adherence to ISO and/or IEEE specifications for math functions in general and floating-point math in particular. In floating-point heavy applications, you might see significant performance improvements, but the trade-off is that you give up compatibility with established standards. In some situations, noncompliant math operations might be acceptable, but you are the only one who can make that determination.

■ **Note** Not all of GCC's optimizations can be controlled using a flag. GCC performs some optimizations automatically and, short of modifying the source code, you cannot disable these optimizations when you request optimization using `-O`.

Processor-Specific Optimizations

Due to the wide variety of processor architectures GCC supports, I cannot begin to cover the processor-specific optimizations in this chapter. Appendix A covers, in detail, all of the processor-specific optimizations you can apply. So curious readers who know the details of their CPUs are encouraged to review the material there. In reality, though, the target-specific switches discussed in Appendix A are not properly optimizations in the traditional sense. Rather, they are options that give GCC more information about the type of system on which the code being compiled will run. GCC can use this additional information to generate code that takes advantage of specific features or that works around known misfeatures of the given processor.

Before starting work on this book, I usually used the (architecture-independent) `-O2` option exclusively, and left it to the compiler to do the right thing otherwise. After writing this book, I have expanded my repertoire, adding some additional options that I have found to be useful in specific cases. My goal is to provide some guidelines and tips to help you select the right optimization level and, in some situations, the particular optimization you want GCC to apply. These can only be guidelines, though, because you know your code better than I do.

Automating Optimization with Acovea

If this book teaches you nothing else, you've learned that together the GCC compilers offer approximately 1.3 zillion options. Trying to find the absolute best set of GCC options to use for a specific application and architecture can be tedious at best; so most people stick with the standard GCC optimization options, invoking selected others they have found to be useful over time. This is something of a shame because it overlooks additional optimization possibilities; but your development time has to be optimized, too.

Scott Ladd's Acovea application (`http://www.coyotegulch.com/products/acovea/index.html`) provides an interesting and useful mechanism for deriving optimal sets of optimization switches, using an evolutionary algorithm that emulates natural selection. Lest this sound like voodoo, let's think about how optimization works in the first place. Optimization applies algorithms that potentially improve various code segments; checks the results; and retains those that result in code improvements. This is conceptually akin to the first step of the natural selection process. Acovea simply automates the propagation of satisfactory optimizations to subsequent optimization passes, which apply additional optimizations and measure the results. Optimizations that do not indeed optimize or improve the code are therefore treated as evolutionary dead ends that are not explored.

GCC experts, random posts on the Internet, and books such as this one all offer a variety of general suggestions for trying to derive "the best" optimizations. Unfortunately, these can be contradictory and can never take into account the content and structure of your specific application. Acovea attempts to address this by enabling the exhaustive, automatic analysis of the performance of applications compiled with an iterative set of optimization options. Its exhaustive analysis also enables you to empirically derive information about the most elusive aspect of GCC's optimization options—their interaction. Invoking one promising optimization option may have significant impact on the performance of other optimization options. Acovea enables you to automatically test all configured GCC options in combination with each other, helping you locate the holy grails of application development, the smallest or fastest compiled version of your application.

Building Acovea

You can download the source code for the latest version of Acovea from links located at `http://www.coyotegulch.com/products/acovea/index.html`. Acovea requires that you build and install two additional libraries before actually building Acovea:

- `coyotl`: A library of various routines used by Acovea, including a specialized random number generator, low-level floating point utilities, a generalized command-line parser, and improved sorting and validation tools.
- `evocosm`: A library that provides a framework for developing evolutionary algorithms.

After building and installing these libraries (in order), using the traditional "unpack, configure, make install" sequence, you can build and install Acovea itself. Downloading and building the libacovea package compiles and installs an application called runacovea in `/usr/local/bin` (by default), which is the master control program for GCC optimization testing using Acovea.

■ **Note** Acovea is only supported on actual Linux and Unix-like systems—some work will be required to get it working under Cygwin, but I'm sure Scott Ladd will appreciate any contributions you'd like to make.

Configuring and Running Acovea

The options that Acovea will test for you are defined in an XML-format configuration file. Template configuration files are available from Scott's site at `http://www.coyotegulch.com/products/acovea/acovea-config.html`. These configuration files are highly dependent on the version of GCC you're using, so make sure you get the configuration files for that version of GCC, and also for your specific class of processor. The following is a sample section of an Acovea configuration file:

```
<?xml version="1.0"?>
<acovea_config>
  <acovea version="5.2.0" />
  <description value="gcc 4.1 Opteron (AMD64/x86_64)" />
  <quoted_options value="false" />
  <prime command="gcc"
        flags="-lrt -lm -std=gnu99 -O1 -march=opteron ACOVEA_OPTIONS
            -o ACOVEA_OUTPUT ACOVEA_INPUT" />
  <baseline description="-O1"
        command="gcc"
        flags="-lrt -lm -std=gnu99 -O1 -march=opteron -o ACOVEA_OUTPUT
            ACOVEA_INPUT" />
  <baseline description="-O2"
        command="gcc"
        flags="-lrt -lm -std=gnu99 -O2 -march=opteron -o ACOVEA_OUTPUT
            ACOVEA_INPUT" />
  <baseline description="-O3"
        command="gcc"
        flags="-lrt -lm -std=gnu99 -O3 -march=opteron -o ACOVEA_OUTPUT
            ACOVEA_INPUT" />
  <baseline description="-O3 -ffast-math"
        command="gcc"
        flags="-lrt -lm -std=gnu99 -O3 -march=opteron -ffast-math
            -o ACOVEA_OUTPUT ACOVEA_INPUT" />
  <baseline description="-Os"
        command="gcc"
        flags="-lrt -lm -std=gnu99 -Os -march=opteron -o ACOVEA_OUTPUT
            ACOVEA_INPUT" />

  <!-- A list of flags that will be "evolved" by ACOVEA (85 for GCC 4.1!) -->
  <flags>
    <!-- O1 options (these turn off options implied by -O1) -->
    <flag type="simple" value="-fno-merge-constants" />
    <flag type="simple" value="-fno-defer-pop" />
    <flag type="simple" value="-fno-thread-jumps" />
    <flag type="enum"
        value="-fno-omit-frame-pointer|-momit-leaf-frame-pointer" />
    <flag type="simple" value="-fno-guess-branch-probability" />
    <flag type="simple" value="-fno-cprop-registers" />
    <flag type="simple" value="-fno-if-conversion" />
     . . ..
      <!-- O2 options -->
      <flag type="simple" value="-fcrossjumping" />
      <flag type="simple" value="-foptimize-sibling-calls" />
      <flag type="simple" value="-fcse-follow-jumps" />
      <flag type="simple" value="-fcse-skip-blocks" />
```

```
      <flag type="simple" value="-fgcse" />
      <flag type="simple" value="-fexpensive-optimizations" />
      <flag type="simple" value="-fstrength-reduce" />
      <flag type="simple" value="-frerun-cse-after-loop" />
      <flag type="simple" value="-frerun-loop-opt" />
     …
      <!-- O3 options -->
     <flag type="simple" value="-fgcse-after-reload" />
     <flag type="simple" value="-finline-functions" />
     <flag type="simple" value="-funswitch-loops" />
     …
     <!-- Additional options -->
     <flag type="simple" value="-ffloat-store" />
      <flag type="simple" value="-fprefetch-loop-arrays" />
      <flag type="simple" value="-fno-inline" />
      <flag type="simple" value="-fpeel-loops" />
      …
      <!-- Tuning options that have a numeric value -->
      <flag type="tuning" value="-finline-limit" default="600" min="100"
            max="10000" step="100" separator="=" />
   </flags>
</acovea_config>
```

■ **Note** Acovea can be used with any of the GCC compilers by specifying the compiler that you want to run in the baseline element's `command` attribute in your Acovea configuration file.

Once you have downloaded and optionally customized your configuration file, use the runacovea application to test GCC with the selected options, as in the following example:

```
runacovea –config config-file-name –input source-file-name
```

■ **Note** By default, Acovea selects optimization options that produce the fastest, highest-performance code, but it can also be instructed to optimize for size by specifying the `–size` option on the runacovea command line.

Executing the runacovea application produces a variety of output as Acovea tests your compiler with permutations of the specified options, culminating in output such as the following:

```
Acovea completed its analysis at 2005 Nov 24 08:45:34

Optimistic options:

                        -fno-defer-pop  (2.551)
                     -fmerge-constants  (1.774)
                    -fcse-follow-jumps  (1.725)
                        -fthread-jumps  (1.822)

Pessimistic options:

                        -fcaller-saves  (-1.824)
                      -funswitch-loops  (-1.581)
                        -funroll-loops  (-2.262)
```

```
          -fbranch-target-load-optimize2  (-2.31)
                               -fgcse-sm  (-1.533)
                     -ftree-loop-ivcanon  (-1.824)
                             -mfpmath=387  (-2.31)
                             -mfpmath=sse  (-1.581)

Acovea's Best-of-the-Best:
gcc -lrt -lm -std=gnu99 -O1 -march=opteron -fno-merge-constants
    -fno-defer-pop -momit-leaf-frame-pointer -fno-if-conversion
    -fno-loop-optimize -ftree-ccp -ftree-dce -ftree-dominator-opts
    -ftree-dse -ftree-copyrename -ftree-fre -ftree-ch -fmerge-constants
    -fcrossjumping -fcse-follow-jumps -fpeephole2 -fschedule-insns2
    -fstrict-aliasing -fthread-jumps -fgcse-lm -fsched-interblock -fsched-spec
    -freorder-functions -funit-at-a-time -falign-functions -falign-jumps
    -falign-loops -falign-labels -ftree-pre -finline-functions -fgcse-after-reload
    -fno-inline -fpeel-loops -funswitch-loops -funroll-all-loops -fno-function-cse
    -fgcse-las -ftree-vectorize -mno-push-args -mno-align-stringops
    -minline-all-stringops -mfpmath=sse,387 -funsafe-math-optimizations
    -finline-limit=600 -o /tmp/ACOVEAA7069796 fibonacci_all.c

Acovea's Common Options:
gcc -lrt -lm -std=gnu99 -O1 -march=opteron -fno-merge-constants
    -fno-defer-pop -momit-leaf-frame-pointer -fcse-follow-jumps -fthread-jumps
    -ftree-pre -o /tmp/ACOVEAAA635117 fibonacci_all.c

-O1:
gcc -lrt -lm -std=gnu99 -O1 -march=opteron -o /tmp/ACOVEA58D74660 fibonacci_all.c

-O2:
gcc -lrt -lm -std=gnu99 -O2 -march=opteron -o /tmp/ACOVEA065F6A10 fibonacci_all.c

-O3:
gcc -lrt -lm -std=gnu99 -O3 -march=opteron -o /tmp/ACOVEA934D7357 fibonacci_all.c

-O3 -ffast-math:
gcc -lrt -lm -std=gnu99 -O3 -march=opteron -ffast-math -o /tmp/ACOVEA408E67B6
fibonacci_all.c

-Os:
gcc -lrt -lm -std=gnu99 -Os -march=opteron -o /tmp/ACOVEAAB2E22A4 fibonacci_all.c
```

As you can see, Acovea produces a listing of the combination of options that produce the best results, as well as information about optimization options that it suggests should be added to each of the standard GCC optimization options.

As mentioned previously, exhaustively testing all GCC optimization options and determining the interaction between them takes a prohibitive amount of time for mere mortals, even graduate students and interns. Acovea is an impressive application that can do this for you, and can help you automagically produce an optimal executable for your application. For more detailed information about using Acovea and integrating it into the make process for more complex applications than the simple example we've used, see `http://www.coyotegulch.com/products/acovea`.

CHAPTER 6

Analyzing Code Produced with GCC Compilers

Chapter 2 discussed the various types of optimizations that GCC's C compiler can perform for you. Some of these optimizations are automatic, whereas others only make sense based on the characteristics of the application you are trying to optimize. For example, you can only decide where and if to use optimizations such as unrolling or inlining loops after studying your application and identifying loops that might benefit from these optimizations. (Unrolling or inlining loops means to insert the code for each iteration of a loop in sequence so that a loop is no longer present; the loop is replaced by a series of sequential statements that explicitly perform the contents of the loop for each value of the variable that controls the loop.)

Determining how to analyze an application can be problematic. The most common mechanism for examining variables and analyzing the sequence and frequency of functions called in an application is the same mechanism generally used during debugging—sprinkling `printf()` function calls throughout your code. Aside from being time-consuming, this approach has the unfortunate side effect of changing application performance. Each `printf()` call has a certain amount of overhead, which can mask truly obscure problems such as timing and allocation conflicts in your applications. The `printf()` mechanism also shows only the sequence of function calls within one run of your application. It does not inherently provide information about the execution of those functions unless you do the calculations yourself and then display the statistics. Just like adding calls to `printf()` in the first place, performing execution time calculations in your own debugging functions can change the behavior that you are trying to observe.

Beyond simply identifying opportunities for optimization, application analysis is important for a variety of other reasons—testing, for instance. One measure of a good test suite is that it exercises as much of the code for your application as possible. In order to devise a good test suite, it is important to be able to identify all of the possible sequences of function calls within your application; you can do so with an item known as a *call graph*. Attempting to do this manually can be tedious even on a good day, and requires continuous updating each time you change your application. Luckily, as explained later in this chapter, you can automatically generate call graphs and examine code coverage using the GNU gcov tool.

Analyzing the behavior, performance, and interaction between function calls in your applications is generally classified into two different, but related, types of analysis:

- *Code coverage analysis*: Shows how well any single run of the application exercises all of the functions in your application.
- *Code profiling*: Provides performance information by measuring the behavior of and interaction between the functions in any single run of your application. There are many different types of profiling—the most common of these is execution profiling, which measures and reports on the time spent in each function during a run of an application.

Luckily (and not too surprisingly), the GNU Compiler Collection (GCC) addresses both of these needs for developers:

- *Code coverage*: GCC compilers include an application and two compilation options that make it easy to perform code analysis. The GNU Coverage application, gcov, automatically creates a call graph for your applications, helping you identify how well any given test code exercises the various execution paths and relationships between the functions in an application. This application produces some general profiling information, but its focus is on coverage analysis.
- *Code profiling*: GCC compilers also provide a number of compilation options that make it easy for you to use code profilers such as the GNU profiler, gprof.

■ **Note** The code coverage and profiling options discussed in this chapter can be used with any of the languages supported by the GNU Compiler Collection. This is a huge advantage for developers who need to develop code in multiple languages, because they can reuse the same knowledge and tips. While the majority of the examples in this chapter are provided in the C programming language, examples in other languages are also occasionally provided to highlight the cross-language nature of all GCC compilers. When reading, please substitute the appropriate concept from the language that you're using.

This chapter explains how to do code coverage analysis using GCC compilation options and gcov, and then discusses code profiling. The section "Test Coverage Using GCC and gcov" provides an overview of test coverage mechanisms, explains gcc compilation options related to test coverage, presents examples of using gcov for coverage analysis, and discusses the coverage-related files produced during compilation and by the gcov application. The section "Code Profiling Using GCC and gprof" explains GCC options related to profiling, and then discusses how to use gprof to identify potential bottlenecks and areas for possible optimization in your applications.

Test Coverage Using GCC and gcov

Test coverage is the measurement of how well any given code or run of an application exercises the code that makes up the application. The next few sections provide an overview of test coverage, highlight the options provided by GCC compilers for test coverage, discuss all of the auxiliary data files produced by those options, and supply examples of using gcov to perform test coverage analysis.

Before you can explore the options provided by GCC compilers for performing coverage analysis, you need an understanding of test coverage basics. In the next section, we give you an overview of the most common types of test coverage and explore situations in which tests must be cleverly constructed to augment the built-in test coverage capabilities provided by GCC compilers.

Overview of Test Coverage

A variety of different approaches to test coverage are commonly used to measure how well certain tests exercise a given application. The most common of these are the following:

- *Statement coverage*: Statement coverage measures whether each statement in an application is exercised. Keep in mind that the number of statements is not necessarily directly related to the number of lines of code, since single lines of code can contain multiple statements, though putting multiple statements on a single line is poor practice from a testing (and readability) point of view. Test output that lists every line of code in an application and whether it is covered can be too voluminous to manage properly. For this reason, most statement coverage testing identifies test coverage in terms of basic blocks of statements that are sequential and non-branching. This "statement coverage shorthand" is usually referred to as *basic block coverage*. As explained later in this chapter, GCC provides built-in support for basic block coverage reporting. Statement coverage and similar approaches are also often referred to as *line coverage*, *block coverage*, and *segment coverage*.
- *Decision coverage*: Decision coverage measures whether all of the possible decisions within an application are being tested with all possible values. Decision coverage is essentially a superset of statement coverage, because in order to exercise all values of all conditional expressions within your code, you have to execute all of the statements within the code for each conditional. Decision coverage is also commonly referred to as *branch coverage* or *all edges coverage*.
- *Path coverage*: Path coverage measures whether all of the possible execution paths within an application are being tested. This usually involves creating a truth table for all of the functions within an application and ensuring that all of the permutations recorded in the truth table are being tested. Complete path coverage is impossible in a single run of an application that has sequential opposing conditionals, as only one or the other can be true. Path coverage is essentially a superset of decision coverage; in order to exercise all possible execution paths in an application, you have to execute all of its code, including all of the code in each conditional, at one time or another. Path coverage is also referred to as *predicate coverage*.
- *Modified condition decision coverage*: Modified condition decision coverage measures whether all of the expressions that lead to each decision or execution path are being tested. For example, modified condition decision coverage measures whether tests cover conditionals predicated upon multiple conditions combined by `and` or `or` statements. Modified condition decision coverage is also referred to as *condition coverage*, *condition-decision coverage*, and *expression coverage*.

All of these coverage metrics examine your code in different ways, and each has its own advocates. Basic block coverage is the type of test coverage that is built-in to GCC, and is the most common test coverage mechanism in general use throughout the industry. By creating detailed test code and test suites, you can use basic block coverage to effectively provide other types of test coverage.

Designing effective test suites is almost an art form, and is certainly a science and discipline with its own rules. To design such test suites requires that you understand how the application you are testing works, and also that you take into account any syntactic peculiarities of the language in which the application you are testing was written. For example, it can be difficult to achieve 100 percent coverage of your code using statement or basic block coverage for applications written in C and C++ for a variety of reasons. Blocks of application code that are designed to catch system error conditions are hard to exercise during testing. Complex single-line statements are similarly difficult to exhaustively test.

Even with well-crafted test suites and applications that are designed to be testable, it is difficult to exercise portions of a robust application that are designed to catch abnormal error conditions that should either never happen or are difficult to induce. An example of a hard-to-induce statement is an error message displayed after using an `exec` call in a standard Linux/Unix `fork()/exec()` statement for spawning a child process from within an application. Since an `exec()` call replaces the code for a running process or thread with an instance of another program, it should never return, and any subsequent statements (usually error messages) should never be seen. An example of this is the `fprintf()` statement in the following block of sample code:

```
child = fork ();
if (child == 0) {
    execvp (program, argument_list);
    fprintf (stderr, "Error \"%s\" in execvp - aborting...\n", strerror());
    abort ();
}
```

In this case, the `fprintf()` statement could only be reached if the `execvp()` call failed, which would only happen if the program could not be found or an external system error condition occurred. It should be difficult for your test code to reproduce improbable errors in your application. If it is not, perhaps they are not so improbable.

Similarly, the C and C++ languages provide a commonly used syntax for embedding decisions within single statements such as the following:

```
result = my_function() ? foo : bar ;
```

In this example, the value of result is `foo` if the call to `my_function()` returns successfully, and `bar` if it does not. In statement or basic block coverage approaches to testing, this type of expression is identified as being tested if it is executed, which does not necessarily say anything about whether the code in `my_function()` is correct.

In general, it is important to note that statement and basic block coverage only measure whether statements are exercised by a set of tests, not whether those statements are logically correct or do what you intend them to do.

A truly good test or set of tests not only executes all of the functions within an application, but also exercises all of the possible execution paths within that application. At first, this seems to be both simple and obvious on the surface, as most of the code in an application consists of lines, sequences of statements, loops, or simple conditionals that look something like the following:

```
if variable1
    foo;
else
    bar;
```

Simple conditionals such as if-then-else clauses or their cousins, the switch statements found in programming languages such as C and C++, are easy enough to exercise and examine. Your tests execute your code, providing an instance of each possible value that is being tested for in the conditional clause, and then, if possible, executes the application with a value that is not being tested for to ensure that any open else clause (or the switch statement's default case) is exercised. This type of test coverage provides both statement coverage, in which each statement is tested and executed, and decision coverage, in which every possible execution path within the code is tested by making sure that every possible decision within the code is executed once.

Additional testing complexities arise when single lines contain multiple decision points and when those decision points contain compound logical statements. In these instances, it can be easy to overlook test cases that exercise every possible execution path through the code. As an example, consider the following block of pseudo-code:

```
if (variable1 and (variable2 or function1())
     foo;
else
     bar;
```

Most test code would consider this block of code to be fully covered if it caused both the `foo` and the `bar` statements to be executed. However, both of these branches could be tested by simply manipulating the values of `variable1` and `variable2`—in other words, without ever executing the function `function1()`. The `function1()` function could not only perform any number of illegal operations, but could also manipulate global variables that would substantially change the execution environment of the rest of the program.

Testing applications is a science unto itself, as is writing applications that are designed with testability in mind. As you will see throughout the rest of this chapter, the GNU Compiler Collection provides excellent built-in support for coverage analysis. For more detailed information about software testing, designing code for testability, and automating software testing, here are a few good references:

- *Automated Software Testing: Introduction, Management, and Performance*, Elfriede Dustin, Jeff Rashka, John Paul (Addison-Wesley, 1999. ISBN: 0-201-43287-0).
- *Software Test Automation: Effective Use of Test Execution Tools*, Mark Fewster, Dorothy Graham (Addison-Wesley, 1999. ISBN: 0-201-33140-3).
- *Software Testing and Continuous Quality Improvement, Second Edition*, William Lewis (CRC Press, 2004. ISBN: 0-849-32524-2).
- *Systematic Software Testing*, Rick D. Craig, Stefan P. Jaskiel (Artech House, 2002. ISBN: 1-580-53508-9).
- *Testing Computer Software, Second Edition*, Cem Kaner, Hung Q. Nguyen, Jack Falk (John Wiley & Sons, 1999. ISBN: 0471358460).

Compiling Code for Test Coverage Analysis

In order to use the gcov test coverage tool to produce coverage information about a run of your application, you must

- Compile your code using a GCC compiler. The gcov application is not compatible with embedded coverage and analysis information produced by any other compiler.
- Use the `-fprofile-arcs` option when compiling your code with a GCC compiler.
- Use the `-ftest-coverage` option when compiling your code with a GCC compiler.
- Avoid using any of the optimization options provided by the GCC compiler you are using. These options might modify the execution order or grouping of your code, which would make it difficult or impossible to map coverage information into your source files.

Compiling source code with the `-fprofile-arcs` option causes GCC compilers to instrument the compiled code, building a call graph for the application and generating a call graph file in the directory where the source file is located. This call graph file has the same name as the original source file, but replaces the original file extension (`c`, `C`, `CC`, `f90`, and so on) with the `.gcno` extension. This file is created at compile time, rather than at runtime. A call graph is a list identifying which basic blocks are called by other functions, and which functions call other functions. The representation of each call from one function to another in a call graph is known as a *call arc*, or simply an *arc*. The .gcno file contains information that enables the gcov program to reconstruct the basic block call graphs and assign source line numbers to basic blocks. For example, compiling the program fibonacci.c with the `-fprofile-arcs` option creates the file fibonacci.gcno in your working directory.

Running an application compiled with `-fprofile-arcs` creates a file with the same name as the original source file but with the `.gcda` extension (GNU Compiler arc data) in the directory where the source file is located. For example, executing the binary program fibonacci that was compiled with the `-fprofile-arcs` option creates the file fibonacci.gcda in your working directory. The file with the `.gcda` extension contains call arc transition counts and some summary information.

■ **Note** The information in a `.gcda` file is cumulative. Compiling your code with the `-fprofile-arcs` option and then running it multiple times will add all of the information about all runs to the existing `.gcda` file. If you want to guarantee that a `.gcda` file only contains profiling and coverage information about a single run on your application, you must delete any existing `.gcda` file before running the application.

Compiling source code with the `-ftest-coverage` option causes GCC compilers to identify and track each basic block within the source file. This information is recorded in the same `.gcno` file produced when using the `-fprofile-arcs` option, and is therefore created at compile time rather than at runtime.

A basic block is a series of sequential statements that must be executed in order and together without any branching. The information recorded in the `.gcno` file makes it possible for gcov to reconstruct program flow and annotate source files with coverage and profiling information.

For more information about `.gcda` and `.gcno` files, see the section "Files Used and Produced During Coverage Analysis," later in this chapter.

Using the gcov Test Coverage Tool

After compiling your source files with the `-fprofile-arcs` and `-ftest-coverage` options, as explained in the previous section, you can run your application normally (usually with some test scenario). This produces the summary information in the `.gcda` file. You can then use gcov to generate coverage information for any of the source files in your application, as well as display the profiling information produced by that run of your code.

When run on any of your source modules, the gcov program uses the information in the `.bb`, `.bbg`, `.gcno`, and `.gcda` files to produce an annotated version of your source file, with the extension `.gcov`. For example, running gcov on the module fibonacci.c produces the annotated source file fibonacci.c.gcov. The annotated source files produced by gcov use rows of hash marks to identify commands in the source file that were not executed, and also provide numeric summaries of the number of times that each line or basic block in the files was executed.

The gcov program provides a number of command-line options that you can use to get information about gcov or specify the information produced by gcov. These command-line options are shown in Table 6-1.

The next section of this chapter shows an example run of an application and gcov, highlighting the files produced and information displayed at each step.

Table 6-1. *Options for the gcov Program*

gcov Option	Description
`-a`, `--all-blocks`	Using either of these options causes gcov to write execution counts for every basic block rather than just the counts for each line. Specifying this option can be very useful if lines in your input program contain multiple basic blocks.
`-b`, `--branch-probabilities`	Using either of these command-line options enables you to see how often each branch in your application is executed during the sample run(s). Specifying either of these options causes gcov to include branch frequency information in the annotated source.gcov file, showing how often each branch in your application is executed. This frequency information is expressed as a percentage of the number of times the branch is taken out of the number of times it is executed. This option also causes gcov to display summary information about program branches to its standard output.
`-c`, `--branch-counts`	Using either of these command-line options in conjunction with the `-b` or `--branch-probabilities` option modifies the annotated branch count information to show branch frequencies as the number of branches executed rather than the percentage of branches executed. These options do not modify the summary information produced to gcov's standard output by the `-b` or `--branch-probabilities` options.
`-f`, `--function-summaries`	Using these options allows you to output summary information about function calls in the specified source file in addition to the summary information about all calls in the file.
`-h`, `--help`	Using these options displays help about using gcov and then exits without analyzing any files.
`-l`, `--long-file-names`	Specifying either of these options when creating annotated gcov output files creates the names of the annotated versions of include files by concatenating the name of the source files in which they were found to the name of the include file, separated by two hash marks. This can be very useful when the same include file is referenced in multiple source files. For example, if the files foo.c and bar.c both include baz.h, running gcov with the `-l` option on the file foo.c would produce the output files foo.c.gcov and foo.c##baz.h.gcov. Similarly, running gcov with the `-l` option on the file bar.c would produce the output files bar.c.gcov and bar.c##baz.h.gcov.
`-n`, `--no-output`	Using these options allows you to specify that you do not want the gcov output file created.

Table 6-1. *Options for the gcov Program (Continued)*

gcov Option	Description
`-o directory|file, --object-directory directory|file`	Using these options specifies either the directory containing the gcov data files, the directory containing the object file, or the path to the object file itself. If these options are not supplied, it defaults to the current directory. This option is useful if you have moved a directory containing instrumented source code and want to produce test coverage information without recompiling.
`-p, --preserve-paths`	Specifying this option causes gcov to preserve full pathname information in the names of its output files, with each directory separator (usually /) replaced by a hash mark (#). This can be quite useful in archiving coverage reports and output files for projects where constituent files are located in different directories.
`-u, --unconditional-branches`	Specifying this option causes gcov to include the branch probability information for unconditional branches.
`-v, --version`	Using these options allows you to display version information about the gcov application you are running and exit without processing any files.

A Sample gcov Session

This section provides an example of using gcov to provide test coverage information for a small sample application that calculates a specified number of values in the Fibonacci sequence. Listing 6-1 shows the main routine for this application, stored in the file fibonacci.c. Listing 6-2 shows one external routine for this application, which is stored in the file calc_fib.c. The application is stored in multiple files in order to illustrate using gcov with applications constructed from multiple source files, rather than from any programmatic necessity.

Listing 6-1. *The Source Code for the Sample fibonacci.c Application*

```
/*
 * Simple program to print a certain number of values
 *  in the Fibonacci sequence.
 */

#include <stdio.h>
#include <stdlib.h>

int calc_fib(int n);

int main(int argc, char *argv[]) {
    int i,n;

    if (argc == 2)
      n = atoi(argv[1]);
    else {
      printf("Usage: fibonacci num-of-sequence-values-to-print\n");
      exit(-1);
    }
```

```
    for (i=0; i < n; i++)
      printf("%d ", calc_fib(i));
    printf("\n");
    return(0);
}
```

Listing 6-2. *The Source Code for the Auxiliary calc_fib.c Function*

```
/*
 * Function that actually does the Fibonacci calculation.
 */

int calc_fib(int n) {
  if (n == 0) {
    return 0;
  } else if (n == 1) {
    return 1;
  } else
    return((calc_fib(n-2) + calc_fib(n-1)));
}
```

Before running gcc or gcov, the contents of the directory containing the source code for the sample application are as follows:

```
$ ls
calc_fib.c  fibonacci.c  Makefile
```

In the real world, you usually create rules in your Makefile that add a target for generating your application with code coverage options enabled. For simplicity's sake, this section shows the manual commands for enabling code coverage.

First, compile the application with the gcc options necessary to produce the basic block and call graph information used by gcov:

```
$ gcc -fprofile-arcs -ftest-coverage fibonacci.c calc_fib.c -o fibonacci
```

After compilation completes, the contents of the sample application directory are as follows:

```
$ ls
calc_fib.c  calc_fib.gcno  fibonacci  fibonacci.c  fibonacci.gcno Makefile
```

This illustrates that the `.gcno` file (for each compiled file) is produced by gcc during compilation. As mentioned earlier in this chapter, this file contains information that will be used by gcov to annotate any source files that you examine, identifying call arcs, basic blocks, and the number of times each statement or basic block is executed.

Next, we will run the sample application, specifying the command-line argument 11 to generate the first 11 numbers in the Fibonacci sequence:

```
$ ./fibonacci 11
```

```
0 1 1 2 3 5 8 13 21 34 55
```

After running the application, the contents of the sample application's source directory are as follows:

```
$ ls
```

```
calc_fib.c     calc_fib.gcno     fibonacci.c        fibonacci.gcno
calc_fib.gcda  fibonacci         fibonacci.gcda     Makefile
```

Running the application creates the `.gcda` files for each of the source files that comprise the sample fibonacci application. These files are automatically produced in the directory where the source code is compiled. If for some reason the directory that contains the application's source code was moved or no longer exists, you would see messages such as the following when running the application:

```
$ ./fibonacci 11
```

```
0 1 1 2 3 5 8 13 21 34 55
arc profiling: Can't open output file /home/wvh/src/fib/fibonacci.gcda.
arc profiling: Can't open output file /home/wvh/src/fib/calc_fib.gcda.
```

After running the application, all of the necessary data files used by gcov are now available, so you can run gcov on a source file to see coverage and summary profiling information:

```
$ gcov fibonacci.c
```

```
File '/usr/include/sys/sysmacros.h'
Lines executed:0.00% of 6
/usr/include/sys/sysmacros.h:creating 'sysmacros.h.gcov'

File 'fibonacci.c'
Lines executed:77.78% of 9
fibonacci.c:creating 'fibonacci.c.gcov'
```

Listing the application's `src` directory now shows that the file fibonacci.c.gcov has been produced by gcov:

```
$ ls
```

```
calc_fib.c     calc_fib.gcno     fibonacci.c       fibonacci.gcda    Makefile
calc_fib.gcda  fibonacci         fibonacci.c.gcov  fibonacci.gcno    sysmacros.h.gcov
```

■ **Note** The file sysmacros.h.gcov was also generated by gcov when annotating fibonacci.c. This is because this file is included in the build process by another include file, but provides macro definitions containing inline functions that must therefore be instrumented.

Listing 6-3 shows the file fibonacci.c.gcov. This file is an annotated version of the fibonacci.c source file in which execution counts and basic block information are displayed. Note that only the fibonacci.c.gcov file is created—no gcov output file is produced for the other source file used in the sample application. You must separately execute gcov on every source file for which you want to display coverage and summary profiling information.

Listing 6-3. *The Output File Produced by gcov for the Source File fibonacci.c*

```
        -:    0:Source:fibonacci.c
        -:    0:Graph:fibonacci.gcno
        -:    0:Data:fibonacci.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:/*
        -:    2: * Simple program to print a certain number of values
        -:    3: *  in the Fibonacci sequence.
        -:    4: */
        -:    5:
        -:    6:#include <stdio.h>
        -:    7:#include <stdlib.h>
        -:    8:
        -:    9:int calc_fib(int n);
        -:   10:
        1:   11:int main(int argc, char *argv[]) {
        -:   12:    int i,n;
        -:   13:
        1:   14:    if (argc == 2)
        1:   15:      n = atoi(argv[1]);
        -:   16:    else {
    #####:   17:      printf("Usage: fibonacci num-of-sequence-values-to-print\n");
    #####:   18:      exit(-1);
        -:   19:    }
       12:   20:    for (i=0; i < n; i++)
       11:   21:      printf("%d ", calc_fib(i));
        1:   22:    printf("\n");
        1:   23:    return(0);
        -:   24:}
```

The output files produced by gcov contain an indented version of any routines in the specified source file. The numbers in the column at the left of the source code indicate the number of times each line was executed.

■ **Note** In profiling output, the string "######" indicates these lines were never executed during this run.

Some amount of summary profiling information is always collected when executing code that has been instrumented during compilation by using any GCC compiler's `-fprofile-arcs` and `-ftest-coverage` options. Using gcov's `-b` (`--branch-probabilities`) option when running gcov on an instrumented source module displays this information and also incorporates it into the annotated filename.c.gcov module produced by gcov. Continuing with the fibonacci.c example, executing gcov with the `-b` option on the fibonacci.c file displays the following output:

```
$ gcov -b fibonacci.c
```

```
File '/usr/include/sys/sysmacros.h'
Lines executed:0.00% of 6
No branches
No calls
/usr/include/sys/sysmacros.h:creating 'sysmacros.h.gcov'
```

```
File 'fibonacci.c'
Lines executed:77.78% of 9
Branches executed:100.00% of 4
Taken at least once:75.00% of 4
Calls executed:66.67% of 6
fibonacci.c:creating 'fibonacci.c.gcov'
```

Note If you haven't compiled your code using `-fprofile-arcs`, gcov will simply abort. If you see the message "Aborted," recompile your code with the `-fprofile-arcs` option.

Listing 6-4 shows the output file fibonacci.c.gcov produces by running gcov with the `-b` option. Note that the output file still identifies basic blocks and provides line execution counts, but now also identifies the line of source code associated with each possible branch and function call. The annotated source code also displays the number of times that branch is executed.

Listing 6-4. *Branch-Annotated Source Code Showing Branch Percentages*

```
        -:    0:Source:fibonacci.c
        -:    0:Graph:fibonacci.gcno
        -:    0:Data:fibonacci.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:/*
        -:    2: * Simple program to print a certain number of values
        -:    3: *  in the Fibonacci sequence.
        -:    4: */
        -:    5:
        -:    6:#include <stdio.h>
        -:    7:#include <stdlib.h>
        -:    8:
        -:    9:int calc_fib(int n);
        -:   10:
function main called 1 returned 100% blocks executed 82%
        1:   11:int main(int argc, char *argv[]) {
        -:   12:    int i,n;
        -:   13:
        1:   14:    if (argc == 2)
branch  0 taken 100% (fallthrough)
branch  1 taken 0%
        1:   15:      n = atoi(argv[1]);
call    0 returned 100%
        -:   16:    else {
    #####:   17:      printf("Usage: fibonacci num-of-sequence-values-to-print\n");
call    0 never executed
    #####:   18:      exit(-1);
call    0 never executed
        -:   19:    }
       12:   20:    for (i=0; i < n; i++)
branch  0 taken 92%
branch  1 taken 8% (fallthrough)
       11:   21:      printf("%d ", calc_fib(i));
call    0 returned 100%
```

```
call    1 returned 100%
        1:   22:    printf("\n");
call    0 returned 100%
        1:   23:    return(0);
        -:   24:}
```

The construction of the branch percentages around the `for` loop in Listing 6-4 and its call to the `printf()` and `calc_fib()` functions is interesting. You can see that the `for` statement was executed 12 times, the last of which it exited, so this line was executed 11 out of 12 times, or 92 percent of the time. Of the 11 times the loop body was executed, the loop exited the single time the test was false, or 100 percent of the time the exit condition was satisfied. The increment operation was executed each time the `for` loop was executed, which was 11 out of 11 times, or 100 percent of the time.

The `printf()` call and its internal call to the `calc_fib()` routine were each executed 11 out of 11 times; so both of these were executed 100 percent of the time. All of these statistics help you get a feel for the execution path that a given test run takes through your application. This not only shows which code is being used, but also helps you identify dead (unused) code, or code paths that you had expected to be taken more frequently. The branch percentages displayed in the annotated source file show the number of times the branch was called divided by the number of times the branch was executed. Using gcov's `-c` option in conjunction with the `-b` option causes the branch counts in the annotated source code to be measured absolutely rather than as a percentage. The output of the command is the same, as shown in the following output sample, but Listing 6-5 shows the difference in the contents of the fibonacci.c.gcov file.

Listing 6-5. *Annotated Source Code Showing Absolute Branch Counts*

```
        -:    0:Source:fibonacci.c
        -:    0:Graph:fibonacci.gcno
        -:    0:Data:fibonacci.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:/*
        -:    2: * Simple program to print a certain number of values
        -:    3: *  in the Fibonacci sequence.
        -:    4: */
        -:    5:
        -:    6:#include <stdio.h>
        -:    7:#include <stdlib.h>
        -:    8:
        -:    9:int calc_fib(int n);
        -:   10:
function main called 1 returned 100% blocks executed 82%
        1:   11:int main(int argc, char *argv[]) {
        -:   12:    int i,n;
        -:   13:
        1:   14:    if (argc == 2)
branch  0 taken 1 (fallthrough)
branch  1 taken 0
        1:   15:      n = atoi(argv[1]);
call    0 returned 1
        -:   16:    else {
    #####:   17:      printf("Usage: fibonacci num-of-sequence-values-to-print\n");
call    0 never executed
    #####:   18:      exit(-1);
```

```
call    0 never executed
        -:   19:    }
       12:   20:    for (i=0; i < n; i++)
branch  0 taken 11
branch  1 taken 1 (fallthrough)
       11:   21:      printf("%d ", calc_fib(i));
call    0 returned 11
call    1 returned 11
        1:   22:    printf("\n");
call    0 returned 1
        1:   23:    return(0);
        -:   24:}
```

If you are interested in summary information about function calls in the specified source file, gcov's `-f` option displays summary information about the lines of source code executed in each function as well as summary information about the complete source file:

```
$ gcov -f fibonacci.c
```

```
Function 'gnu_dev_major'
Lines executed:0.00% of 2

Function 'gnu_dev_minor'
Lines executed:0.00% of 2

Function 'gnu_dev_makedev'
Lines executed:0.00% of 2

Function 'main'
Lines executed:77.78% of 9

File '/usr/include/sys/sysmacros.h'
Lines executed:0.00% of 6
/usr/include/sys/sysmacros.h:creating 'sysmacros.h.gcov'

File 'fibonacci.c'
Lines executed:77.78% of 9
fibonacci.c:creating 'fibonacci.c.gcov'
```

If the source file had contained more than one routine, the function summary information would have provided statistical information about each function.

As mentioned earlier, using gcov to perform coverage analysis on source files that include other source files (typically `.h` files containing definitions, macros, or inline functions) produces an annotated source.gcov file for the main file and for any included file that contains inline functions. If you are performing coverage analysis on multiple source files from a single application, this can be confusing if the same include file is referenced in multiple source files. Since the whole idea of include files is that they facilitate modular development and compilation by being included in multiple source files, this is a fairly common occurrence in multifile applications.

When analyzing files that include other source files, you can use gcov's `-l` (or `--long-names`) option to cause gcov to produce annotated include files whose names also contain the name of the source file in which they were included. For example, if the files foo.c and bar.c both include baz.h, which includes an inline function declaration, running gcov with the `-l` option on the file foo.c would produce the output files foo.c.gcov and foo.c##baz.h.gcov. Similarly, running gcov with the `-l` option on the file bar.c would produce the output files bar.c.gcov and bar.c##baz.h.gcov.

Files Used and Produced During Coverage Analysis

The gcov application uses two files for each source file that contains executable code and is involved in compilation for coverage analysis. These files have the same basename as the source files they are associated with, except that the original file extension is replaced with the `.gcno` and `.gcda` extensions. These files are always created in the same location as the source files that they are associated with. Neither of these files is designed to be read or directly used without access to the data structure and entry information in the source and include files for gcov.

The `.gcno` file is generated when a source file is compiled by a GCC compiler if the `-fprofile-arcs` or `-ftest-coverage` options are specified. This file contains a list of the program flow arcs (possible branches taken from one basic block to another) for each function in the main source file or any included source file, and also contains information that enables the gcov program to reconstruct the basic function call graphs and assign source line numbers to functions.

The `.gcda` file, created when you run an application that was compiled with the `-fprofile-arcs` option, contains information about function calls and execution paths in one or more runs of the application. One `.gcda` file is created for each source file compiled with the `-fprofile-arcs` option. Once one or more `.gcda` files have been created by a run of your application, subsequent runs of your application append information to this file. To guarantee that a `.gcda` file contains information about a single run of your application, you should delete any existing `.gcda` file before running your application.

For detailed information about the internal structure of these files, see the gcov information in the GNU GCC documentation.

Code Profiling Using GCC and gprof

As discussed in this chapter's introduction, profiling an application determines how often each portion of the application executes and how it performs each time it executes. The goal of most application profiling work is to identify the portions of your application that are the most resource intensive, either computationally or in terms of memory consumption. Once you have this information, you can then look for ways to improve the performance of those functions or your entire application in those areas.

The gcov application, discussed in the first half of this chapter, is designed to provide general profiling information that tells you how often each basic block (sequence of uninterruptible, sequential statements) executes during one or more test runs of the application.

The GNU gprof application, designed for profiling, provides much more detailed analysis of your application and the functions it executes internally than gcov can provide. The GNU gprof application was inspired by the BSD Unix prof application (also found in many System III and System V Unix variants). Because GCC compilers are widely used on a variety of systems, the gprof application provides options to produce output in the file formats used by prof, in case prof is still supported and used on your computer's operating system.

The GNU gprof application provides several forms of profiling output:

- *A flat profile*: Shows the amount of time your application spends in each function, the number of times each function is called, the amount of time spent in profiling-related overhead in each function, and an estimation of the margin of error in the profiling data.
- *A call graph*: Shows the relationships between all of the functions in your application. The call graph shows the calling sequence for the functions in your application, and also shows how much time is spent in each function and any other functions that it calls.
- *An annotated source code listing*: Shows the number of times each line in the program's source code is executed.

As with gcov, you can add the internal code to your application to produce profiling data by specifying a variety of options. These options tell the GCC compiler to automatically instrument each function so that your application produces profiling data that is automatically written to external files that can be displayed and analyzed by gprof.

As you will see later in this section, the profiling options that are automatically available to you when using GCC compilers also enable you to write your own profiling routines. These routines will be inserted in your compiled code, which will be executed at the entry and exit points of each function in the application. This enables you to extend and customize the profiling functionality you get for free with the GCC compiler package. Not a bad deal.

Obtaining and Compiling gprof

Unlike gcov, gprof is not part of the standard GCC distribution. The gprof application is a part of the binutils package, which is a collection of GNU utilities for managing and manipulating binary object files. The binutils package includes the GNU Linker (ld), the GNU Assembler (as), the GNU Library Archiver (ar) and its companion indexer (ranlib), the addr2line utility for mapping binary addresses to lines of source code, the objcopy and objdump utilities for converting and dumping object files, and many other GNU/Linux favorites.

The process of compiling and installing the binutils package is much the same as with any other GNU package:

1. Download the latest version of binutils from `http://sources.redhat.com/binutils`.
2. Extract the contents of the source archive using a command such as the following:

   ```
   tar zxvf binutils-2.16.1.tar.gz
   ```

3. Change your working directory to the directory produced in the previous step, as in this example:

   ```
   cd binutils-2.16.1
   ```

4. Run the configure script to configure the archive for your system and to create the Makefiles for the various libraries and utilities in the package:

   ```
   ./configure
   ```

5. Build all of the utilities in the package by using a single top-level make command:

   ```
   make
   ```

6. If there were no errors in the previous step, install the updated utilities, related libraries, and associated man pages and other online information using the following two-level make command as the root user:

   ```
   make install
   ```

By default, this sequence of commands will install the updated version of gprof (and the other utilities in the binutils package) in `/usr/local/bin`. Make sure this directory is located in your path before `/usr/bin`, the traditional installed location of the gprof utility, or you may accidentally run an older version of gprof. You can determine the gprof binary that appears first in your path by executing the `which gprof` command. You can then determine the version of that copy of gprof by executing the command `gprof -v`.

Compiling Code for Profile Analysis

In order to cause your application to produce profiling information that can subsequently be analyzed by gprof, you must

- Compile your code using a GCC compiler. The gprof application is not inherently compatible with embedded profiling and analysis information produced by any other compiler.
- Use the `-pg` option when compiling your code with your GCC compiler to activate and link the profiling libraries.

■ **Note** If you are using a GCC compiler on a Solaris system or other system that provides the older Unix prof application, you can produce profiling information suitable for analysis using prof by compiling your code with the `-p` option rather than the `-pg` option.

- (Optional) Use the `-g` option if you want to do line-by-line profiling. This standard debugging option tells the compiler to preserve the symbol table in the compiled code, but also inserts the debugging symbols that gprof (and gdb) use to map addresses in the executable program back to the lines of source code they were generated from.
- (Optional) Use the `-finstrument-functions` option if you have written your own profiling functions to be called when each instrumented function is entered and before each function returns to the calling routine. Using this option and writing profiling functions is discussed in the section "Adding Your Own Profiling Code Using GCC's C Compiler."

During initial profiling, you also want to avoid using any of the optimization options provided by your GCC compiler. These options might modify the execution order or grouping of your code, which would make it difficult or impossible to identify areas for optimization.

Running an application compiled with the `-pg` option creates the file gmon.out when your program exits normally. Unlike the source.gcda coverage information file produced by gcov, the gmon.out file is recreated each time you run your application and only contains information about a single run of the application. Unlike the output files produced by gcov, this file is produced in the working directory of your program upon exiting. For example, programs that use the `chdir()` function will create the gmon.out file in the last directory used. Similarly, the `chroot()` function sends the gmon.out file to a directory relative to the new root directory. If there already is a file by the name of gmon.out in the directory where your program exists, its contents will be overwritten with the new profiling information.

■ **Note** Because the gmon.out file is produced when your program exits normally (by returning from the `main()` function or by calling the `exit()` function), programs that crash or are terminated by a signal will not create the gmon.out file. Similarly, the gmon.out file will not be produced if your program exits by calling the low-level `_exit()` function.

Versions of GCC prior to GCC 3.2 used the `-a` option to generate basic block profiling information, writing this information to a file named bb.out. This option and the output file it produced are obsolete—this functionality is now provided by using gcov to produce an annotated source code listing after compiling your code with the basic block coverage options discussed in the section "Compiling Code for Test Coverage Analysis."

Using the gprof Code Profiler

After compiling your source files with the options discussed in the previous section, you can run your application normally (usually with some test scenario). This produces profiling information in the gmon.out file. You can then use gprof to display profiling information in a number of different ways. The gprof application can even use the information in the gmon.out file to suggest automatic optimizations, such as function ordering in your executable, or link ordering during the linking phase of compiling your application.

The next section of this chapter shows an example run of an application and gprof, highlighting the types of information you may commonly want to produce and examine using gprof.

The gprof program provides a huge number of command-line options you can use to specify the types of information produced by gprof or the types of analysis it performs on the data in the gmon.out file.

Symbol Specifications in gprof

Some of the command-line options supported by gprof enable you to restrict the output of gprof to specific functions, any functions used in a specific file, or any functions used in a specific line of a specific file. These are known as symbol specifications. The syntax of a symbol specification can be one of the following:

- *filename*: Causes gprof to produce profiling output for any function called in the specified source file. For example, setting fibonacci.c as a symbol specification causes gprof to produce profiling output only for functions called in the file fibonacci.c.
- *function-name*: Causes gprof to produce profiling output only for any function in the source code with the specified name. For example, setting the symbol specification to calc_fib causes gprof to produce profiling output only for the function `calc_fib()` and any functions that it calls. If you have multiple functions with the same name (e.g., global and local functions), you can specify a function from a particular source file by using the notation *source-file*:*function-name*.
- *filename*:*line-number*: Causes gprof to produce profiling information only for functions called from the specified line in the specified source file.

Table 6-2 shows the command-line options provided by gprof.

Table 6-2. *Command-Line Options for the gprof Program*

Option	Description
`-a`, `--no-static`	Suppress the printing of profiling information for functions that are not visible outside the file in which they are defined. For example, profiling an application that is partially composed of a source file that contains definitions of the functions `foo()` and `bar()`, where `bar()` is only called by `foo()`, will only report calls to the function `foo()` in profiling output. Any time spent in these "local" functions will be added to the profiling output for the function(s) that calls them in gprof's flat profile and call graph output.
`-A`[*symbol-specification*], `--annotated-source`[=*symbol-specification*]	Cause gprof to print annotated source code. If a symbol specification is provided, annotated source output is only generated for functions that match that specification. In order to print annotated source code, you must have used the `-g` option when compiling your application, so that mandatory symbol information is present in the profiled binary.
`-b`, `--brief`	Cause gprof not to print the explanations of the flat profile and call graph. Though useful, once you are familiar with this information, it is helpful to be able to suppress it in your output because it is also incredibly verbose.
`-c`, `--static-call-graph`	Cause gprof to provide extra analysis of your application's call graph, listing functions present in the source code but never actually called. Symbol table entries for these functions must be present—for example, if you link with an object file that is not compiled with profiling support, any functions are present in that object file but not called by your application will not be displayed unless that object file was compiled with `-g` and `-pg`. Using these options provides a quick and easy way to identify dead code, which is code that is never used and which you may be able to remove.
`-C`[*symbol-specification*], `--exec-counts`[=*symbol-specification*]	Print summary information about the functions in your sample application and the number of times they are called. Providing a symbol specification restricts the summary information to functions matching the symbol specification.
`-d`[*num*], `--debug`[=*num*]	Specify a debugging level. If no debug level is specified, these options display all debugging information. Debugging options are rarely used unless you suspect a problem with gprof, in which case the maintainer may request that you provide output using a specific debugging level.

Table 6-2. *Command-Line Options for the gprof Program (Continued)*

Option	Description
`--demangle[=style]`, `--no-demangle`	(*Used only with C++ applications*) Specify whether the symbol names in compiled C++ code will be decoded from simple low-level names into user-level names, including class information, enabling gprof to differentiate between them in its output. The optional style argument enables you to specify a demangling style, based on your compiler. Possible styles are auto, which causes gprof to automatically select a demangling style based on examining your program's source code; gnu, the default value for code compiled using g++; lucid, which specifies that gprof should demangle symbol names based on the algorithm used by the Lucid C++ compiler's name encoding algorithm; and arm, which results in gprof demangling function names using the algorithm specified in *The Annotated C++ Reference Manual*, Margaret A. Ellis and Bjarne Stroustrup (Addison-Wesley, 1990. ISBN: 0-201-51459-1).
`-D`, `--ignore-non-functions`	Ignore symbols that are not known to be functions. These options are only supported on certain systems, such as Solaris and HP-UX systems.
`--file-ordering` *map_file*	Prints a suggested order in which object files in the application should be linked. This ordering is based on profiling data that shows which functions call which other functions, and can improve paging and cache behavior in the compiled application. This option is typically used with the `-a` option in order to suppress linking information about external functions that are contained in libraries and whose link order you therefore have no control over.
`--function-ordering`	Prints a suggested order in which functions in the program should be organized to improve paging and cache behavior in the compiled application.
`-i`, `--file-info`	Print summary information about the number of histogram records, the number of records in the call graph, and the number of basic block count records in the gmon.out file, and exit.
`-I` *dirs*, `--directory-path=`*dirs*	Specify a colon-separated list of directories that should be searched for source files related to the application.
`-J[`*symbol-specification*`]`, `--no-annotated-source[=`*symbol-specification*`]`	Suppress printing of annotated source code. If a symbol specification is provided, an annotated source code listing is printed that does not include symbols that match the specification. In order to print annotated source code, you must have used the `-g` option when compiling your application, so that mandatory symbol information is present in the profiled binary.
`-k` *from/to*	Deletes specified arcs from the call graph. Arcs are specified as from/to pairs, where *from* is the function calling a second function, and *to* is the function being called.

Table 6-2. *Command-Line Options for the gprof Program (Continued)*

Option	Description
`-l`, `--line`	Enable line-by-line profiling, which causes histogram and call graph profiling information to be associated with specific lines in the application's source code. Line-by-line profiling is much slower than basic block profiling, and can magnify statistical inaccuracies in gprof output.
`-L`, `--print-path`	Display the full pathname of source files when doing line-by-line profiling.
`-m` *num*, `--min-count=`*num*	Suppress information about symbols that were executed less than *num* times. These options are only meaningful in execution count output.
`-n`[*symbol-specification*], `--time`[`=`*symbol-specification*]	Restrict the symbols whose values are propagated in gprof's call graph analysis to those that match the symbol specification.
`-N`[*symbol-specification*], `--no-time`[`=`*symbol-specification*]	The inverse of the `-n` option, cause gprof not to propagate call graph data for symbols matching the symbol specification.
`-O`*name*, `--file-format=`*name*	Specify the format of the gmon.out data file. Valid values for *name* are *auto* (the default), *bsd*, *4.4bsd*, and *magic*.
`-p`[*symbol-specification*], `--flat-profile`[`=`*symbol-specification*]	Display flat profile information. If a symbol specification is specified, gprof only prints flat profile information for symbols matching that symbol specification.
`-P`[*symbol-specification*], `--no-flat-profile`[`=`*symbol-specification*]	Suppress printing flat profile information. If a symbol specification is provided, gprof prints a flat profile that excludes matching symbols.
`-q`[*symbol-specification*], `--graph`[`=`*symbol-specification*]	Display call graph analysis information. If a symbol specification is provided, gprof only prints call graph information for symbols matching that symbol specification and for children of those symbols.
`-Q`[*symbol-specification*], `--no-graph`[`=`*symbol-specification*]	Suppress printing call graph analysis. If a symbol specification is provided, gprof prints a call graph that excludes symbols matching that symbol specification.
`-s`, `--sum`	Summarize profile data information and write it to the file gmon.sum. By using the `-s` option and specifying gmon.sum as a profiling data file on the gprof command line, you can merge the data from multiple profile data files into a single summary file and then use a gprof command line such as gprof *executable* gmon.sum to display the summary information.
`-T`, `--traditional`	Display output in traditional BSD (Berkeley Standard Distribution) style.
`-v`, `--version`	Display the version number of gprof and then exit without processing any data.
`-w` *width*, `--width=`*width*	Set the width of the lines in the call graph function index to *width*.

Table 6-2. *Command-Line Options for the gprof Program (Continued)*

Option	Description
`-x`, `--all-lines`	Annotate each line in annotated source output with the annotation for the first line in the basic block to which it belongs. In order to print annotated source code, you must have used the `-g` option when compiling your application, so that mandatory symbol information is present in the profiled binary.
`-y`, `--separate-files`	Cause annotated source output to be written to files with the same name as each source file, appending -ann to each filename. In order to print annotated source code, you must have used the `-g` option when compiling your application, so that mandatory symbol information is present in the profiled binary.
`-z`, `--display-unused-functions`	List all functions in the flat profile, even those that were never called. You can use these options with the `-c` option to identify functions that were never called and can potentially therefore be eliminated from the source code for your application.
`-Z`[*symbol-specification*], `--no-exec-counts`[=*symbol-specification*]	Suppress printing summary count information about all the functions in your application and the number of times they were called. If a symbol specification is provided, summary count information is printed that excludes symbols matching the symbol specification.

■ **Note** In addition to using command-line options, you can affect the behavior of gprof by setting the `GPROF_PATH` environment variable to a list of directories in which gprof should search for application source code files, if you are using gprof options that produce annotated source code listings.

A Sample gprof Session

This section walks through an example of using gprof to provide profiling information for the same small application used in the gcov discussion of this chapter. This program calculates a specified number of values in the Fibonacci sequence. The sample source code for this application is shown in Listings 6-1 and 6-2, earlier in this chapter.

Before running gcc or gprof, the contents of the directory containing the source code for the sample application are as follows:

```
$ ls
```

```
calc_fib.c  fibonacci.c  Makefile
```

First, compile the application using gcc's `-pg` option to integrate profiling information in the binary:

```
$ gcc –pg –g fibonacci.c calc_fib.c -o fibonacci
```

After compilation completes, the contents of the sample application directory are as follows:

```
$ ls
```

```
calc_fib.c  fibonacci  fibonacci.c  Makefile
```

Next, run the sample application, specifying the command-line argument 11 to generate the first 11 numbers in the Fibonacci sequence:

```
$ ./fibonacci 11
```

```
0 1 1 2 3 5 8 13 21 34 55
```

After running the application, you get the following as the contents of the sample application's source directory:

```
$ ls
```

```
calc_fib.c  fibonacci  fibonacci.c  gmon.out  Makefile
```

Running the application creates the gmon.out file, which contains profiling information based on the run of the application. You can now run gprof to produce information about the behavior and performance of the application. Running gprof in its simplest form on our sample application, gprof fibonacci (gprof fibonacci.exe for Cygwin users), produces the output shown in Listing 6-6. The default output of gprof is somewhat verbose, but contains a good deal of useful information about interpreting the output you receive.

Listing 6-6. *Default Output from gprof*

```
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00       11     0.00     0.00  calc_fib

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.
```

```
calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.

                Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) no time propagated

index % time    self  children    called     name
                                     442             calc_fib [1]
                0.00    0.00       11/11           main [8]
[1]      0.0    0.00    0.00       11+442      calc_fib [1]
                                     442             calc_fib [1]
-----------------------------------------------

 This table describes the call tree of the program, and was sorted by
 the total amount of time spent in each function and its children.

 Each entry in this table consists of several lines.  The line with the
 index number at the left hand margin lists the current function.
 The lines above it list the functions that called this function,
 and the lines below it list the functions this one called.
 This line lists:
     index  A unique number given to each element of the table.
            Index numbers are sorted numerically.
            The index number is printed next to every function name so
            it is easier to look up where the function in the table.

     % time This is the percentage of the 'total' time that was spent
            in this function and its children.  Note that due to
            different viewpoints, functions excluded by options, etc,
            these numbers will NOT add up to 100%.

     Self   This is the total amount of time spent in this function.

    children This is the total amount of time propagated into this
            function by its children.

     Called This is the number of times the function was called.
            If the function called itself recursively, the number
            only includes non-recursive calls, and is followed by
            a '+' and the number of recursive calls.
```

```
     Name   The name of the current function.  The index number is
            printed after it.  If the function is a member of a
            cycle, the cycle number is printed between the
            function's name and the index number.

 For the function's parents, the fields have the following meanings:

     Self   This is the amount of time that was propagated directly
            from the function into this parent.

   children This is the amount of time that was propagated from
            the function's children into this parent.

     Called This is the number of times this parent called the
            function '/' the total number of times the function
            was called.  Recursive calls to the function are not
            included in the number after the `/'.

     Name   This is the name of the parent.  The parent's index
            number is printed after it.  If the parent is a
            member of a cycle, the cycle number is printed between
            the name and the index number.

 If the parents of the function cannot be determined, the word
 '<spontaneous>' is printed in the `name' field, and all the other
 fields are blank.

 For the function's children, the fields have the following meanings:

     self   This is the amount of time that was propagated directly
            from the child into the function.

   children This is the amount of time that was propagated from the
            child's children to the function.

     called This is the number of times the function called
            this child '/' the total number of times the child
            was called.  Recursive calls by the child are not
            listed in the number after the '/'.

     name   This is the name of the child.  The child's index
            number is printed after it.  If the child is a
            member of a cycle, the cycle number is printed
            between the name and the index number.

 If there are any cycles (circles) in the call graph, there is an
 entry for the cycle-as-a-whole.  This entry shows who called the
 cycle (as parents) and the members of the cycle (as children.)
 The '+' recursive calls entry shows the number of function calls that
 were internal to the cycle, and the calls entry for each member shows,
 for that member, how many times it was called from other members of
 the cycle.

Index by function name

   [1] calc_fib
```

As shown in the preceding output listing, the default output of gprof provides several types of information:

- *Flat profile information*: Shows the number of times each function was called and the amount of time spent in each call. In the case of the sample application, this shows that the `calc_fib()` routine was called 11 times from the fibonacci program's main routine, and that it ran so quickly it appears to have taken no time. This is because profiling measurements are expressed in milliseconds but summaries are shown in 1/10 seconds—the program did not require enough execution time to round up to 1 millisecond.
- *Call graph information*: Shows how often each function was called and the functions from which it was called. This shows that the `calc_fib()` routine was called 11 times from the program's main routine, but that the `calc_fib()` routine was called by itself 442 times (since it is a recursive routine).
- *A function index*: Summarizes all of the functions in the application, indexed by function name to make it easy to correlate the index to the flat profile and call graph output.

You can display just the summary information by executing gprof with the `-b` (`brief`) option, as in the following example:

```
$ gprof -b fibonacci
```

```
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00       11     0.00     0.00  calc_fib

                     Call graph

granularity: each sample hit covers 2 byte(s) no time propagated

index % time    self  children    called     name
                                  442            calc_fib [1]
                0.00    0.00      11/11          main [8]
[1]      0.0    0.00    0.00      11+442     calc_fib [1]
                                  442            calc_fib [1]
-----------------------------------------------

Index by function name

   [1] calc_fib
```

This example provides the basic profiling information without the explanation of any of the fields that are part of gprof's default output.

Displaying Annotated Source Code for Your Applications

As shown in the previous section, gprof offers a tremendous number of options, not all of which are necessary to show here. However, one of gprof's most useful options is the `-A` option, which displays annotated source code for the functions in the application, marked up with profiling information.

In order to display annotated source code from gprof, complete symbol information must be present in the binary, so the program must be compiled using a GCC compiler with both the profiling (`-pg`) and the symbol debugging (`-g`) switches, as in the following example:

```
$ gcc -pg -g calc_fib.c fibonacci.c -o fibonacci
```

After executing the fibonacci program to generate the gmon.out profiling file, you can run gprof with the `-A` option to display annotated source code for the only function in the application, `calc_fib()`, as shown in the following example:

```
$ ./fibonacci 11
```

```
0 1 1 2 3 5 8 13 21 34 55
```

```
$ gprof -A fibonacci
```

```
*** File /home/wvh/writing/books/GCC/second/fib_src/external/calc_fib.c:
                /*
                 * Function that actually does the Fibonacci calculation
                 */

          11 -> int calc_fib(int n) {
                #ifdef DEBUG
                  printf("calc_fib called with %d\n", n);
                #endif
                  if (n == 0) {
                    return 0;
                  } else if (n == 1) {
                    return 1;
                  } else
                    return((calc_fib(n-2) + calc_fib(n-1)));
                }

Top 10 Lines:
     Line      Count
        5         11

Execution Summary:

        1   Executable lines in this file
        1   Lines executed
   100.00   Percent of the file executed

       11   Total number of line executions
    11.00   Average executions per line
```

If you want to see the complete, annotated source code with profiling information for a simple application, you can combine it into a single source module, as shown in Listing 6-7.

Listing 6-7. *Single Source Module for the Sample Application*

```
/*
 * Simple program to print a certain number of values
 *  in the Fibonacci sequence.
 */

#include <stdio.h>
#include <stdlib.h>

static int calc_fib(int n) {
  if (n == 0) {
    return 0;
  } else if (n == 1) {
    return 1;
  } else
    return((calc_fib(n-2) + calc_fib(n-1)));
}

int main(int argc, char *argv[]) {
    int i,n;

    if (argc == 2)
      n = atoi(argv[1]);
    else {
      printf("Usage: fibonacci num-of-sequence-values-to-print\n");
      exit(-1);
    }
    for (i=0; i < n; i++)
      printf("%d ", calc_fib(i));
    printf("\n");
    return(0);
}
```

Compiling this module and then using the `gprof -A fibonacci` command to display the annotated source code listing will present the entire listing, as shown in Listing 6-8.

Listing 6-8. *Annotated Source Code Listing of the Single Source Module*

```
*** File /home/wvh/writing/books/GCC/second/fib_src/integrated/fibonacci.c:
         /*
          * Simple program to print a certain number of values
          *  in the Fibonacci sequence.
          */

         #include <stdio.h>
         #include <stdlib.h>

   11 -> static int calc_fib(int n) {
           if (n == 0) {
             return 0;
           } else if (n == 1) {
             return 1;
```

```
          } else
            return((calc_fib(n-2) + calc_fib(n-1)));
        }

##### -> int main(int argc, char *argv[]) {
            int i,n;

            if (argc == 2)
              n = atoi(argv[1]);
            else {
              printf("Usage: fibonacci num-of-sequence-values-to-print\n");
              exit(-1);
            }
            for (i=0; i < n; i++)
              printf("%d ", calc_fib(i));
            printf("\n");
            return(0);
        }
```

```
Top 10 Lines:

     Line      Count

        9         11

Execution Summary:

        2   Executable lines in this file
        2   Lines executed
   100.00   Percent of the file executed

       11   Total number of line executions
     5.50   Average executions per line
```

Note If you compile your source code without using the `-g` option, it will return a message like the following on a Linux system when you try to display annotated source code output using gprof:

```
$ gprof -A fibonacci
gprof: could not locate '/home/wvh/src/glibc-2.2.5/csu/gmon-start.c'
```

This message is somewhat misleading, as it identifies the location of the source code for the compiler's monitoring routines, based on the location of this source file when gcc was compiled on your system. Recompiling your application with the `-g` option will eliminate this problem by correctly identifying the entry points that are specific to your application.

In conjunction with the profiling options provided by all GCC compilers, the gprof application can give you substantial insights into the execution of your application and the amount of time spent in each of its routines. The gprof application can be quite useful as a guide to optimization, but cannot directly help with debugging. However, gcc has one more profiling card up its sleeve—the ability to

automatically insert user-defined code at the entry and exit points of every function in your application via the `-finstrument-functions` compilation option. As explained in the next section, this gcc option can provide the link between profiling and debugging that you may find useful when testing and optimizing your applications.

Adding Your Own Profiling Code Using GCC's C Compiler

The gcc compiler's `-finstrument-functions` option automatically inserts a call to two profiling functions that will automatically be called just after entering each function and just before returning from each function. A call to the `__cyg_profile_func_enter()` function is inserted after entering each function, and a call to the `__cyg_profile_func_exit()` function is made just before returning from each function. The prototypes for these functions are shown here:

```
void __cyg_profile_func_enter (void *this_fn,
                               void *call_site);
void __cyg_profile_func_exit  (void *this_fn,
                               void *call_site);
```

The parameters to these functions are the address of the current function and the address from which it was called, as provided in the application's symbol table.

Mapping Addresses to Function Names

You can display the values of these parameters from within these functions using the standard `printf()` or `fprintf()` `%p` format string. You can then use the GNU addr2line function (part of the binutils package that also provides gprof) to translate the address to the line in a specified source file from which it was called, as in the following example:

```
$ addr2line -e fibonacci3inst 0x80484b4
```

```
/home/wvh/writing/books/GCC/second/fib_src/integrated/fibonacci.c:22
```

Note The sample output fragments in this section are just examples—your output will be different from, but similar to, what appears here.

The addr2line program's `-e` option specifies the name of the executable that contains the specified address. The output from the addr2line command shows the source file and the number of the line of source code in that file that maps to the specified address.

In order to use the addr2line application, you must have preserved symbolic debugging information in the application by compiling it with the `-g` option. If you forget to use the `-g` option when compiling, the addr2line application displays a message like the following:

```
$ addr2line -e fibonacci3inst 0x80484b4
```

```
/home/wvh/src/glibc-2.2.5/csu/init.c:0
```

Because this executable does not contain symbolic debugging information, the addr2line function cannot find the specified address, and therefore returns a pointer to one of the GNU C library's

initialization routines. This is the same sort of problem discussed in the previous section when trying to use gprof to display annotated source code for applications that are not compiled with symbolic debugging information.

Simply defining the `__cyg_profile_func_enter()` and `__cyg_profile_func_exit()` functions in one of the existing source files for your application will enable you to add your own code to these functions in order to extract additional profiling or debugging data beyond that which is automatically provided by gprof. Adding calls to these functions obviously slows the performance of your application, but should do so consistently in all instrumented functions. It is also simple enough to set up your Makefile so that you can easily compile your application without the `-finstrument-functions` option whenever you want to measure pure performance.

Common Profiling Errors

The most common error made when implementing the profiling functions provided by gcc is to simply add them to one of the source files you are compiling with the `-finstrument-functions` option. By default, this will cause these functions themselves to be instrumented, which will cause your application to crash, because the `__cyg_profile_func_enter()` function will recursively call itself until a stack overflow occurs. You can eliminate this problem in either of two ways:

- Put the code for these two functions in a separate source module that you do not compile with the `-finstrument-functions` option. This is the simplest approach, but requires special handling of this source file in your Makefile, which is easy to forget or overlook.
- Use the `no_instrument_function` attribute on the entry and exit functions to specify that these functions should not be instrumented.

Listing 6-9 shows the complete source code for the sample fibonacci.c application used throughout this section, with sample code for the profiling functions inserted that shows the syntax required to set the `no_instrument_function` attribute.

Listing 6-9. *Sample Application Showing Manual Profiling Calls*

```
/*
 * Simple program to print a certain number of values
 *  in the Fibonacci sequence.
 */

#include <stdio.h>
#include <stdlib.h>

void  __attribute__((__no_instrument_function__))
      __cyg_profile_func_enter(void *this_fn, void *call_site)
{
 printf(" Entered function %p, called from %p\n", this_fn, call_site);
}

void __attribute__((__no_instrument_function__))
     __cyg_profile_func_exit(void *this_fn, void *call_site)
{
 printf("Exiting function %p, called from %p\n", this_fn, call_site);
}
```

```
static int calc_fib(int n) {
  if (n == 0) {
    return 0;
  } else if (n == 1) {
    return 1;
  } else
    return((calc_fib(n-2) + calc_fib(n-1)));
}

int main(int argc, char *argv[]) {
    int i,n;

    if (argc == 2)
      n = atoi(argv[1]);
    else {
      printf("Usage: fibonacci num-of-sequence-values-to-print\n");
      exit(-1);
    }
    for (i=0; i < n; i++)
      printf("%d ", calc_fib(i));
    printf("\n");
    return(0);
}
```

■ **Note** Setting attributes only works within separately compiled code units. If you want to set attributes globally, you should put the setting in a header file that is included in each separately compiled block of code.

Some sample output from a run of this application, compiled with the `-finstrument-functions` option, is shown here:

```
Entered function 0x804852a, called from 0x40052657
Entered function 0x80484b4, called from 0x804859d
Exiting function 0x80484b4, called from 0x804859d
0 Entered function 0x80484b4, called from 0x804859d
Exiting function 0x80484b4, called from 0x804859d
1 Entered function 0x80484b4, called from 0x804859d
Entered function 0x80484b4, called from 0x80484fa
Exiting function 0x80484b4, called from 0x80484fa
Entered function 0x80484b4, called from 0x8048508
...
```

When combined with the compilation options required for test coverage analysis and profiling, the `-finstrument-functions` option is a final, powerful arrow in your debugging, analysis, and profiling quiver. Calling user-specified profiling code can save you time in the debugger and can also make it easy to integrate applications compiled with gcc with any third-party or in-house profiling capabilities that you may already be using.

CHAPTER 7

Using Autoconf and Automake

Beyond simply being a robust, high-powered multiuser operating system written in a popular programming language, much of the appeal of Unix lies in the promise of portability. But as soon as the split between the AT&T and the BSD variants of Unix occurred, portability became more like a possible goal than an actual reality.

One way of resolving portability issues is to use platform-specific definitions throughout your code, leading to source code that may contain as many conditionals (`#ifdef YOUR-UNIX-VARIANT`) as actual lines of source code. Though often unreadable, this is a workable solution, except that it requires the maintainer of a portable application to be aware of the nuances of every possible system on which people might want to use that application.

A more workable solution is to test dynamically the characteristics of both your hardware and operating system and generate appropriate Makefiles that are customized for your platform. The applications discussed in this chapter, Autoconf and Automake, do exactly that. Autoconf is an application that generates shell scripts that automatically configure source code packages. Automake is a companion application that can automatically generate Makefiles that are compliant with the GNU Coding Standards.

This chapter explains the division of labor between Autoconf and Automake and surveys the applications they use in order to simplify the configuration process. Sections of this chapter demonstrate how to install and configure Automake and Autoconf, and how to use Autoconf and Automake to make it easy to distribute application source code, so it can be painlessly configured and compiled on a wide variety of Unix, Linux, and other Unix-like operating systems.

Introducing Unix Software Configuration, Autoconf, and Automake

As mentioned in the introduction to this chapter, one of the original selling points of Unix was that its applications were easily ported from one Unix system to another. This was largely due to the fact that they were written in the C programming language, which was born on Unix systems and is still the default programming language of all Unix and Unix-like systems.

However, simply using the same programming language does not guarantee as much as one might hope. As Unix became more widely used, differences between Unix implementations began to be more pronounced, especially between the AT&T-based SYSIII and SYSV flavors of Unix and academically inspired versions of Unix, most of which were based on one version or another of the Berkeley Standard Distribution, commonly known as BSD. Variations in underlying subsystems, such as networking, were one of the major differences between the two, but other differences became more pronounced and more widespread as time passed. By the mid-1980s, versions of Unix such as AIX, AOS, DG/UX, HP-UX, IRIX, SunOS, SYSV, Ultrix, USG, VENIX, and XENIX were among the many flavors of Unix that existed.

Unfortunately, many of these Unix variants used different application programming interfaces (APIs) and different libraries of related functions, ranging from basic data manipulation, such as sorting, all the way to system-level APIs for networking, file and directory manipulation, and so on. A common approach to portability in C source code is to use conditional compilation via `#ifdef` symbols, which use system-defined symbols that identify the operating system version and platform on which the application is being compiled. Although conditionals are necessary and still used today, including them for each Unix system that is available could make the source code for even the simplest C application unreadable.

Conditionalizing code for operating system–dependent features is a pain because many Unix variants share the same sets of features and libraries, leading to a significant amount of duplication within `#ifdef`s. Portable Unix applications therefore began to move toward `#ifdef`s that were based on sets of features rather than specific operating system versions whenever possible. Rather than system-specific statements such as `#ifdef AIX`, a `#ifdef` such as `#ifdef HAS_BCOPY` could be used to differentiate between classes of Unix and Unix-like systems.

Feature-based portability and the generally increasing complexity of setting `#ifdef`s and maintaining applications that were portable across multiple types of Unix systems cried out for a programmatic solution. One of the earliest attempts along these lines was the metaconfig program by Larry Wall, Harlan Stenn, and Raphael Manfredi, which produces interactive shell scripts named configure that are still used to configure applications such as Perl. Other significant attempts at cross-platform, single-source configuration are the configure scripts used by Cygnus Solutions (a major open source/GNU pioneer that was later acquired by Red Hat), gcc, and David MacKenzie's autoconf program.

FUN WITH CONFIGURE SCRIPTS

The early configure scripts produced by the metaconfig program were not only useful, but also entertaining. While testing the parameters and configuration of your system, they also offered some humorous insights, such as my personal favorite:

```
Congratulations! You're not running Eunice...
```

Eunice was a Unix emulation layer that ran on VMS systems. Though useful because it opened up VMS systems to the world of freely available Usenet applications and enabled Unix users to get real work done on VMS systems almost immediately, the whole notion was somewhat unholy.

In a somewhat rare gesture of application solidarity in the Unix environment, David MacKenzie's autoconf program was the eventual winner of the Unix/Linux configuration sweepstakes because it provided a superset of the capabilities of its brethren, with the exception of the interactive capabilities of the metaconfig program.

When he began writing autoconf, MacKenzie was responsible for maintaining many of the GNU utilities for the Free Software Foundation (FSF), and needed a standard way to configure them for compilation on the variety of platforms on which they were used. The vast number of GNU utilities and possible platforms was a strong motivation to write a powerful and flexible configuration utility.

The autoconf program provides an easily extended set of macros that are written for use with the Unix m4 macro processor. Macros are single statements that are subsequently expanded into many other statements by a macro processor. The m4 macro processor is the most common macro processor used on Unix systems, but most programming languages also provide a macro processor, such as the multistage macro preprocessor named cpp that is used with the C programming language.

The autoconf program invokes the m4 macro processor to generate a shell script named configure that a user then executes to customize and configure application source code for a particular computer

system. The autoconf program uses a single input file, configure.ac, (formerly known as configure.in), which contains specific macro statements that provide information about an application and the other applications and utilities required to compile it. Complex applications that use autoconf may have multiple configure.ac files in various subdirectories, each of which defines the configuration requirements for the library or distinct functionality provided by the source code located in that directory.

Unfortunately, identifying the location and names of relevant include files, supporting libraries, and related applications located on a specific system is only half the battle of successfully compiling an application on a particular platform. Since the dawn of Unix time, most Unix applications have been compiled using a rule-driven application called make, which invokes compilers and other utilities actually to produce an executable from all the components of a program's source code. The make program does the following:

- Enables developers to identify relationships and dependencies between the source modules, include files, and libraries that are required for successful compilation.
- Specifies the sequence in which things must be compiled in order to build an application successfully.
- Avoids recompilation by limiting compilation to the portions of an application that are affected by any change to source code or statically linked libraries.

The make program was originally written by Stu Feldman at AT&T for an early version of Unix. As a part of AT&T Unix, the source code for the make program was not freely available, and has therefore largely been replaced by the GNU make program. GNU make provides all of the features of the original make program and a host of others. Other versions of the make program are still active, such as the bmake program used by most modern BSD-inspired versions of Unix (for example, FreeBSD and NetBSD), and the imake program used by the X Window System project; but GNU make is probably the most common version of the program used today. Subsequent references to the make program throughout this chapter therefore refer to GNU make.

The rules used by any make program are stored in text files named Makefile, which contain dependency rules and the commands necessary to satisfy them. As you might expect, the Makefiles for complex applications are themselves extremely complex as well as platform-specific, since they need to invoke platform-specific libraries and utilities that may be located in different directories or have different names on different systems.

To simplify creating Makefiles, David MacKenzie also developed the automake program, which uses a simple description of the build process that is employed to generate Makefiles. The automake program was quickly rewritten in Perl by Tom Tromey, who still maintains and enhances it. The automake program generates a Makefile for an application from an application-specific input file named makefile.am. A makefile.am file contains sets of make macros and rules that are expanded into a file called makefile.in, which is then processed by the autoconf program to produce a Makefile that is tailored for a specific system.

Like the autoconf program's configure.ac files, a complex application that uses automake may have multiple makefile.am files in various subdirectories and reflect specific requirements for the Makefile used to compile the library or distinct functionality provided by the source code located in that directory. The Makefiles produced by the automake program are fully compliant with the Free Software Foundation's GNU Makefile conventions, though these files are only readable by experts or people who do not get migraine headaches. For complete information about the GNU Makefile conventions, see `http://www.gnu.org/prep/standards/html_node/Makefile-Conventions.html`, which is a portion of the larger GNU Coding Standards document at `http://www.gnu.org/prep/standards.html`.

The last component of a truly platform-independent compilation, build, and execution environment is a tool that facilitates linking to libraries of functions, such as the generic C language

libraries, that are standard on a given platform and are also external to the application you are building. Linking is fairly simple when static libraries are actually linked to the executable you are compiling, but becomes complex when shared libraries are being used. The GNU libtool application automatically handles identifying and resolving library names and dependencies when compiling an application that uses shared libraries. We will cross that bridge when we come to it—Chapter 8 explains the use of this application during the compilation and build process.

The remainder of this chapter shows you how to compile and install autoconf and automake, and how to create the configuration files required to use them to create portable, flexible distributions of your applications that can quickly and easily be compiled on a variety of platforms.

Installing and Configuring autoconf and automake

As you might hope, utilities such as autoconf and automake take advantage of their own capabilities in order to simplify setup. This section explains where to obtain the latest versions of autoconf and automake, and how to configure, compile, and install them.

Deciding Whether to Upgrade or Replace autoconf and automake

The default location for installing autoconf and automake as described in this chapter is `/usr/local/bin`. This will leave any existing versions of autoconf, automake, and related applications on your system. In order to make sure you execute the new versions rather than any existing ones, you will have to verify that the `/usr/local/bin` directory appears in your path before the directory in which the default version of autoconf is located (usually `/usr/bin`).

Both autoconf and automake are configured using configure scripts. You can use the configure script's `--prefix` option to change the base location for all the files produced when compiling either package. For example, executing the configure script with the following command will configure the autoconf or automake Makefile to install binaries in `/usr/bin`, or install architecture-independent data such as info files in `/usr/share`, and so on.

```
$ ./configure --prefix=/usr
```

For information on other options for the configure script, see Table 7-6, which appears in the section "Running Configure Scripts" later in this chapter.

■ **Note** You may want to execute the command `which autoconf` in order to determine which version of autoconf your system will use; if you want to use versions that you have already installed in `/usr/local/bin`, adjust the prefix accordingly.

If you are using a system that installs and manages software packages using the RPM Package Manager (RPM), overwriting existing binaries and associated files will cause problems if you subsequently upgrade your system or an affected package. You can always use the rpm command's `--force` option to force removal of packages whose binaries have been overwritten.

In general, simply modifying your path to make sure you are executing the "right" version of an application is simpler than overwriting existing binaries, and it makes it easier for you to fall back to the "official" version if you discover incompatibilities in a new version and simplifies system upgrades of package-based Linux systems. The downside of this approach is that you have to make sure every user of the system updates their path so that they execute the updated version of an application.

The examples in this section configure, build, test, and install the autoconf and automake binaries in their default locations, the appropriate subdirectories of `/usr/local`.

Building and Installing autoconf

autoconf comes preinstalled with most Linux distributions for which you have installed a software development environment for the C and C++ programming languages. However, if you are writing software that you want to distribute and you want to be able to take advantage of the latest fixes and enhancements to autoconf, it can never hurt to get the latest version. autoconf is a relatively stand-alone package that you can safely upgrade without risking the introduction of any incompatibilities on your system. The home page for the autoconf utility is at `http://www.gnu.org/software/autoconf`.

autoconf depends on the GNU m4 macro preprocessor, which is installed by default on most Linux systems. The home page for GNU m4 is at `http://www.gnu.org/software/m4/`, though its primary development site, from which you can download the latest version, is at `http://savannah.gnu.org/projects/m4/`. The current version of m4 at the time of this writing was 1.4.3.

If the Perl interpreter is installed on your system when you are configuring, building, and installing autoconf from source code, the configure script will configure the Makefile so that a number of Perl scripts associated with autoconf are configured for your system. These auxiliary utilities are then installed as part of the `make install` command described later in this section. A complete list of the scripts and auxiliary utilities that can be installed as part of the autoconf package is shown in Table 7-1, later in this chapter.

You can obtain the latest version of autoconf from the download page (`http://ftp.gnu.org/gnu/autoconf`). At the time of this writing, the latest version of autoconf was 2.59—this is the version that we discuss in this chapter.

MAILING LISTS FOR AUTOCONF

If you are very interested in Autoconf or you want to report a bug, discuss its use, or make suggestions for future improvements, the Free Software Foundation hosts several Autoconf-related mailing lists through the `http://www.gnu.org/` site:

- The autoconf list is a forum for asking questions, answering them, discussing usage, or suggesting and discussing new features for Autoconf. You can subscribe to or unsubscribe from this list at `http://mail.gnu.org/mailman/listinfo/autoconf/`.
- The autoconf-patches mailing list contains the latest patches to Autoconf and is where you should submit your changes if you have found and fixed a bug or implemented a new feature in Autoconf. You can subscribe to or unsubscribe from this list at `http://mail.gnu.org/mailman/listinfo/autoconf-patches/`.

You can download a gzipped source file for the latest version of autoconf using your favorite browser or through a faster command-line tool such as wget. For example, you can download the archive containing the latest autoconf source using the command

```
$ wget http://ftp.gnu.org/gnu/autoconf/autoconf-2.59.tar.gz
```

Once you have downloaded the gzipped source archive, you can extract its contents using a standard `tar` command such as the following:

```
$ tar zxvf autoconf-2.59.tar.gz
```

This command creates the directory `autoconf-2.59`. To begin building autoconf, change to this directory and execute the configure script. In a textbook example of bootstrapping, the configure script was produced by autoconf when the maintainers packaged up this latest version using autoconf.

The following is an example of the output from the configure script when run on a sample Linux system:

```
$ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking for expr... /usr/bin/expr
checking for gm4... no
checking for gnum4... no
checking for m4... /usr/bin/m4
checking whether m4 supports frozen files... yes
checking for perl... /usr/bin/perl
checking for emacs... emacs
checking whether emacs is sufficiently recent... yes
checking for emacs... emacs
checking where .elc files should go... ${datadir}/emacs/site-lisp
configure: creating ./config.status
config.status: creating config/Makefile
config.status: creating tests/Makefile
config.status: creating tests/atlocal
config.status: creating tests/autoconf
config.status: creating tests/autoheader
config.status: creating tests/autom4te
config.status: creating tests/autoreconf
config.status: creating tests/autoscan
config.status: creating tests/autoupdate
config.status: creating tests/ifnames
config.status: creating man/Makefile
config.status: creating lib/emacs/Makefile
config.status: creating Makefile
config.status: creating doc/Makefile
config.status: creating lib/Makefile
config.status: creating lib/Autom4te/Makefile
config.status: creating lib/autoscan/Makefile
config.status: creating lib/m4sugar/Makefile
config.status: creating lib/autoconf/Makefile
config.status: creating lib/autotest/Makefile
config.status: creating bin/Makefile
config.status: executing tests/atconfig commands
```

Once you have configured the autoconf source code for your system, you can build autoconf by simply executing the `make` command in the directory `autoconf-2.59`.

```
$ make
 [long, verbose output not shown]
```

Once autoconf has compiled successfully, you should execute the `make check` command to execute the tests provided with autoconf. These have the side effect of ensuring that the version of m4 provided on your system is fully compatible with the requirements of the version of autoconf that you have just compiled:

```
[wvh@64bit:autoconf-2.59]$ make check
Making check in bin
make[1]: Entering directory `/home/wvh/src/autoconf-2.59/bin'
make[1]: Nothing to be done for `check'.
make[1]: Leaving directory `/home/wvh/src/autoconf-2.59/bin'
Making check in tests
make[1]: Entering directory `/home/wvh/src/autoconf-2.59/tests'
make  autoconf autoheader autoreconf autom4te autoscan autoupdate ifnames
make[2]: Entering directory `/home/wvh/src/autoconf-2.59/tests'
make[2]: `autoconf' is up to date.
make[2]: `autoheader' is up to date.
make[2]: `autoreconf' is up to date.
make[2]: `autom4te' is up to date.
make[2]: `autoscan' is up to date.
make[2]: `autoupdate' is up to date.
make[2]: `ifnames' is up to date.
make[2]: Leaving directory `/home/wvh/src/autoconf-2.59/tests'
make  check-local
make[2]: Entering directory `/home/wvh/src/autoconf-2.59/tests'
./autom4te --language=autotest -I . suite.at -o testsuite.tmp
mv testsuite.tmp testsuite
/bin/sh ./testsuite
## ---------------------------- ##
## GNU Autoconf 2.59 test suite. ##
## ---------------------------- ##

Executables (autoheader, autoupdate...).

  1: Syntax of the shell scripts                ok
  2: Syntax of the Perl scripts                 ok
  3: autom4te cache                             ok
  4: autoconf --trace: user macros              ok
  5: autoconf --trace: builtins                 ok
  6: autoconf: forbidden tokens, basic          ok
  7: autoconf: forbidden tokens, exceptions     ok
  8: ifnames                                    ok
  9: autoheader                                 ok
 10: autoupdate                                 ok
 11: autoupdating AC_LINK_FILES                 ok
 12: autoupdating AC_PREREQ                     ok
[additional verbose output not shown]
```

As you can see from this sample output, autoconf provides a fairly exhaustive set of tests to verify its functionality. autoconf executes a number of different sets of tests, including testing the standard executables (shown in the sample output), m4, specific sets of macros for the different languages supported by GCC, and interoperability with related software such as libtool. If your new version of autoconf passes all the tests, you can install it by becoming the root user and using the standard `make install` command:

```
$ su root
Password: root-password
# make install
```

As you read in the section "Deciding Whether to Upgrade or Replace autoconf and automake," the autoconf package is configured to install binaries and supporting files into subdirectories of the directory `/usr/local`. This section details information about changing the location where autoconf installs, and the pros and cons of doing so.

If the Perl interpreter is found on your system when you configure autoconf, the autoconf package builds and installs several related utilities. Table 7-1 lists these applications and explains what each is used for. Using these applications is discussed later in this chapter, in the section "Configuring Software with autoconf and automake."

Table 7-1. *Applications in the autoconf Package*

Utility	Description
autoconf	The shell script that invokes m4 to generate the configure script from the description of your application's configuration in the configure.ac file.
autoheader	A Perl script that parses the source files associated with your application and extracts information about the header files used by the application. This script is only generated and installed if Perl is found on your system when you configure the autoconf package.
autom4te	A Perl script that provides a wrapper for configuring the m4 environment, loading m4 libraries, and so on. This script is only generated and installed if Perl is found on your system when you configure the autoconf package.
autoreconf	A Perl script that generates new distribution files if the files in the GNU build system have been updated. This script is only generated and installed if Perl is found on your system when you configure the autoconf package.
autoscan	A Perl script that generates an initial configure.ac for you by scanning the contents of an application source directory and making a best guess about dependencies and compilation requirements. This script is only generated and installed if Perl is found on your system when you configure the autoconf package.
autoupdate	A Perl script used to update an autoconf configuration file (typically configure.ac) to reflect any changes in autoconf macro syntax or macro names. This script is only generated and installed if Perl is found on your system when you configure the autoconf package.
ifnames	A Perl script that parses C source files and extracts a list of any conditionals (`#if`, `#elif`, `#ifdef`, and `#ifndef`) used in those files. This script is only generated and installed if Perl is found on your system when you configure the autoconf package.

You are now ready to use the new version of autoconf, as described in the section "Configuring Software with autoconf and automake," or to build automake, as described in the next section.

Obtaining and Installing Automake

Like Autoconf, Automake is preinstalled with most Linux distributions in which you have installed a software development environment for the C and C++ programming languages. However, if you are writing software that you want to distribute and you want to be able to take advantage of the latest fixes and enhancements to Automake, it can never hurt to get the latest version. Automake is pretty much a stand-alone package that you can safely upgrade without risking your system. Automake has

a standard page at the GNU site (`http://www.gnu.org/software/automake`), but its actual home page is at the site (`http://sourceware.org/automake`), where Tom Tromey, the author of the Perl version and its maintainer, works. You can get the latest, most cutting-edge versions of Automake there.

Note Automake depends on you having a recent version of the Perl interpreter on your system. You will usually have Perl installed on a system you are using for development, since many development-related utilities are written in Perl. However, if you need or want to install Perl on your system, you can download source code or binary versions of it from the official Perl home page at `http://www.perl.com/pub/a/language/info/software.html`. The examples in this chapter were tested using Perl 5.6, which was the major version provided with most Linux distributions at the time of this writing.

You can obtain the latest version of automake from the download page at the GNU site (`ftp://ftp.gnu.org/gnu/automake`). At the time of this writing, the latest version of automake was 1.9.5—this is the version discussed in this chapter.

MAILING LISTS FOR AUTOMAKE

If you want to become involved with the Automake community or are interested in learning more about it, both Red Hat and the Free Software Foundation host Automake-related mailing lists:

- The automake-cvs and automake-prs lists provide commit messages from CVS and status messages regarding problem reports tracked in the GNATS database where automake issues are recorded. These lists are both hosted at Red Hat. You can subscribe to either of these lists by using the subscription form located on the Automake page at `http://sources.redhat.com/automake`.
- The automake list is a forum for asking questions, answering them, discussing usage, or suggesting and discussing new features for Automake. You can subscribe to or unsubscribe from this list at `http://mail.gnu.org/mailman/listinfo/automake`.
- The bug-automake list is the place you should report any bugs you discover in automake. You can subscribe to or unsubscribe from this list at `http://mail.gnu.org/mailman/listinfo/bug-automake`.
- The automake-patches mailing list contains the latest patches to Automake. This is where you should check if you are waiting for a bug fix for the current version of Automake. You can subscribe to or unsubscribe from this list at `http://mail.gnu.org/mailman/listinfo/automake-patches`.

You can download the archive containing the latest automake source using the following command:

```
$ wget ftp://ftp.gnu.org/gnu/automake/automake-1.9.5.tar.gz
```

Once you have downloaded the gzipped source archive, extract the contents using a standard `tar` command:

```
$ tar zxvf automake-1.9.5.tar.gz
```

This command creates the directory `automake-1.9.5`. To begin building automake, change to this directory and execute the configure script found there.

The following is an example of the output from the configure script when run on a sample Linux system:

```
$ ./configure
checking build system type... x86_64-unknown-linux-gnu
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking for perl... /usr/bin/perl
checking for tex... tex
checking whether autoconf is installed... yes
checking whether autoconf works... yes
checking whether autoconf is recent enough... yes
checking whether ln works... yes
checking for grep that handles long lines and -e... /usr/bin/grep
checking for egrep... /usr/bin/grep -E
checking for fgrep... /usr/bin/grep -F
configure: creating ./config.status
config.status: creating Makefile
config.status: creating doc/Makefile
config.status: creating lib/Automake/Makefile
config.status: creating lib/Automake/tests/Makefile
config.status: creating lib/Makefile
config.status: creating lib/am/Makefile
config.status: creating m4/Makefile
config.status: creating tests/Makefile
config.status: creating tests/defs
config.status: creating tests/aclocal-1.9
config.status: creating tests/automake-1.9
```

Once configuration has completed, you can build an installable version of automake by simply typing **make** at the command prompt. Because Automake is written in Perl, its build phase is essentially a query-replace operation that inserts the location of the Perl binary on your system into the automake and aclocal scripts that are the two executables produced and installed from the automake package. The speed of Automake's configuration and build steps is more than made up for by the time required to run the vast number of tests that are provided with Automake.

To run the tests provided with Automake to verify its correctness, execute the command `make check`. The following is some sample output from running the Automake tests:

```
$ make check
Making check in .
make[1]: Entering directory `/home/wvh/src/automake-1.9.5'
make[1]: Nothing to be done for `check-am'.
make[1]: Leaving directory `/home/wvh/src/automake-1.9.5'
Making check in doc
make[1]: Entering directory `/home/wvh/src/automake-1.9.5/doc'
make[1]: Nothing to be done for `check'.
make[1]: Leaving directory `/home/wvh/src/automake-1.9.5/doc'
Making check in m4
make[1]: Entering directory `/home/wvh/src/automake-1.9.5/m4'
make[1]: Nothing to be done for `check'.
make[1]: Leaving directory `/home/wvh/src/automake-1.9.5/m4'
Making check in lib
make[1]: Entering directory `/home/wvh/src/automake-1.9.5/lib'
Making check in Automake
```

```
make[2]: Entering directory `/home/wvh/src/automake-1.9.5/lib/Automake'
Making check in tests
make[3]: Entering directory `/home/wvh/src/automake-1.9.5/lib/Automake/tests'
make  check-TESTS
make[4]: Entering directory `/home/wvh/src/automake-1.9.5/lib/Automake/tests'
PASS: Condition.pl
PASS: DisjConditions.pl
PASS: Version.pl
PASS: Wrap.pl
==================
All 4 tests passed
==================
[Much additional output deleted]
```

Automake provides more than 430 tests to verify its functionality in a variety of ways. Once these tests are completed, you can install Automake using the standard `make install` command. As with Autoconf, the Automake package is configured to install into subdirectories of the directory `/usr/local`. For information about changing the location where automake installs and the pros and cons of doing so, see the previous section "Deciding Whether to Upgrade or Replace autoconf and automake."

Configuring Software with autoconf and automake

Once you have autoconf, automake, and other utilities from those packages installed on your system, you can actually use them to configure an application. This section explains how to create the configuration and build description files used by autoconf and automake to automate the configuration, compilation, installation, and packaging of your application(s).

At a minimum, using autoconf and automake requires that you create two files: a configuration description file for autoconf that contains the basic m4 macros autoconf needs to generate a configure script for your application, and a build description file that Automake uses to generate a Makefile. You must also create a few auxiliary files that must exist in order for automake to create a packaged distribution of your application conforming to the basic requirements of an application that can be configured using the GNU build tools.

The following sections explain how to create these files and the sequence in which you must execute autoconf, automake, and the configure script itself to generate an appropriate Makefile for your system and configuration. For convenience, these sections use the same small fibonacci application that appears in Chapter 6 to provide a real example of creating configuration files. Listings 6-1 and 6-2 show the C source code for the sample application. To review briefly, the file fibonacci.c contains the `main` routine for the application, and calls the `calc_fib()` routine that is contained in the calc_fib.c source file. This simple application has no external dependencies other than the stdio.h and stdlib.h include files, and has no platform-specific function or structure declarations.

Creating Configure.ac Files

There are two standard approaches to creating the configuration file used by autoconf:

- Use the autoscan utility to generate an initial configuration file you fine-tune to satisfy your application's build configuration requirements.
- Create a simple autoconf configuration description file and then manually enhance it to reflect any additional configuration requirements you discover.

The configuration file used by modern versions of the autoconf script is configure.ac.

■ **Note** Historically, the name of the configuration description file used by the autoconf program has always been configure.in. The `.in` suffix in this filename has led to some confusion with other input files used by the configure script, such as config.h.in, so the "proper" name of the configuration file used by autoconf is now configure.ac; the `.ac` suffix indicates that the file is used by autoconf rather than its offspring. If you have files named configure.ac and configure.in in your working directory when you run autoconf, the autoconf program will process the file configure.ac rather than the file configure.in.

If you have Perl installed on your system, building and installing the autoconf package produces a Perl script called autoscan. By default, the autoscan script examines the source modules in the directory from which it is executed and generates a file called configure.scan. If a configure.ac file is already located in the directory from which autoscan is executed, the autoscan script reads the configure.ac file and will identify any omissions in the file (typically, by exiting with an error message).

As an example, the source directory for the sample fibonacci application looks like the following:

```
$ ls
calc_fib.c  fibonacci.c  run_tests
```

Executing the autoscan application and relisting the contents of this directory shows the following files:

```
$ autoscan
$ ls
autoscan.log  calc_fib.c  configure.scan  fibonacci.c  run_tests
```

If you already have a configure.ac file, the autoscan.log file will contain detailed messages about any advanced macros autoscan believes are required by your application but are not present in the configure.ac file. If you do not already have a configure.ac file or there are no apparent configuration problems, the autoscan.log file produced by a run of autoscan will be empty.

■ **Note** If you do not have a configure.ac file and are using autoscan to generate a template, the autoscan program will exit with an error code of 1, but will still create the template configure.scan file.

The configure.scan file produced by running the autoscan program for the sample application looks like the following:

```
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.

AC_PREREQ(2.59)
AC_INIT(FULL-PACKAGE-NAME, VERSION, BUG-REPORT-ADDRESS)
AC_CONFIG_SRCDIR([fibonacci.c])
AC_CONFIG_HEADER([config.h])
# Checks for programs.
AC_PROG_CC
# Checks for libraries.
# Checks for header files.
AC_HEADER_STDC
AC_CHECK_HEADERS([stdlib.h])
```

```
# Checks for typedefs, structures, and compiler characteristics.
# Checks for library functions.
AC_OUTPUT
```

To use this with the sample programs shown in Chapter 6, you need to remove the line referring to config.h, since the sample program doesn't use that include file, and add the following line near the end of the file telling it to create a Makefile:

```
AC_CONFIG_FILES([Makefile])
```

The autoconf macros in the autogenerated file are defined in Table 7-2.

Table 7-2. *Entries in the Sample autoconf File*

Entry	Description
AC_PREREQ	A macro produced by (and therefore requiring) version 2.59 of the Autoconf package.
AC_INIT	The basic autoconf initialization macro, which requires three arguments: a user-defined name for the application package being configured by autoconf, the version of the package that is currently being configured, and an e-mail address to which bugs in the application should be reported.
AC_CONFIG_SRCDIR	A macro that takes the name of a source file that must be present in the source directory in order to configure and compile the application. This macro is essentially just a sanity check for autoconf. Its value is typically the name of the file containing the main routine for the application you are configuring.
AC_CONFIG_HEADER	A macro that specifies that you want to use a central include file containing cross-platform or multiplatform configuration information. The default name of this configuration include file is config.h. If your application does not depend on platform-specific capabilities, you can delete this macro entry—it is inserted by default by the autoscan script so you won't overlook this feature if you want to take advantage of it.
AC_PROG_CC	A macro that checks for the existence of a C compiler.
AC_HEADER_STDC	A macro that checks for the existence of ANSI C header files by testing for the existence of the header files stdlib.h, stdarg.h, string.h, and float.h.
AC_CHECK_HEADERS	A macro that checks for a specific C language header file other than stdio.h.
AC_CONFIG_FILES	A macro that causes autoconf to create the specified file from a template file, performing variable substitutions as needed based on the other macro statements in the autoconf configuration file. By default, the template file has the same name as the specified file, but with an `.in` extension; you can specify another template file by using a colon to separate the name of the template file from the name of the file that is to be generated. For example, Makefile:foo.in specifies that the Makefile should be produced from a template file named foo.in.
AC_OUTPUT	A macro that causes autoconf to generate the shell script config.status and execute it. This is typically the last macro in any autoconf configuration file.

■ **Note** Just as in the C programming language, the values passed to autoconf macros are enclosed within parentheses. The preprocessors associated with programming languages have their own argument-parsing routines, which compare argument lists against the prototypes defined for each function. Because autoconf uses the m4 macro preprocessor to expand keywords into more complex entries in your Makefile and may need to make multiple passes over your input files, arguments to Autoconf macros are often enclosed within square brackets, which tell m4 where each argument begins and ends. Each pass of m4 strips off one set of square brackets. If you are trying to pass an argument to an Autoconf macro and Autoconf is not processing it correctly, you may need to enclose that argument within square brackets. This is especially true for arguments that contain whitespace. The autoscan program automatically inserts sets of square brackets around all the arguments passed to the Autoconf macros that it generates, just to be on the safe side.

Once you have generated a template configuration file using autoscan, you should copy or rename this file to the default autoconf configuration file configure.ac. At this point, you can delete any unnecessary statements and modify any lines that contain default values. Continuing with our example, the modified configure.ac file would look like the following:

```
AC_PREREQ(2.59)
AC_INIT(Fibonacci, 1.0, vonhagen@vonhagen.org)
AC_CONFIG_SRCDIR([fibonacci.c])
AC_PROG_CC
AC_HEADER_STDC
AC_CHECK_HEADERS([stdlib.h])
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

The modified file specifies appropriate values for the `AC_INIT` macro, removes comments, and removes unnecessary macros such as (in this case) `AC_CONFIG_HEADER`.

■ **Note** The autoconf macros in this sample configure.ac file are a small subset of all of the autoconf macros that are available. The documentation for autoconf (provided in GNU info format) is the ultimate reference for the list of currently supported autoconf macros, since it reflects changes to autoconf that may have occurred since this book was published. Using autoconf's `-v` option when processing your configure.ac file displays a quick list of the macros supported by your version of autoconf.

At this point, you can actually run autoconf to generate an initial configure script, which should complete successfully if no typos exist in your configure.ac file.

```
$ autoconf
$ ls
autom4te.cache  calc_fib.c  configure  configure.ac  fibonacci.c
```

Running autoconf produces a configure script and a directory named `autom4te.cache`. This directory contains cached information about your application that is used by the autoconf package's autom4te application, which provides a high-level layer over m4.

Once you successfully create a configure script, you can try to execute it to watch it do its magic, as in the following output:

```
$ ./configure
checking for gcc... gcc
checking for C compiler default output... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ANSI C... none needed
checking how to run the C preprocessor... gcc -E
checking for egrep... grep -E
checking for ANSI C header files... yes
checking for sys/types.h... yes
checking for sys/stat.h... yes
checking for stdlib.h... yes
checking for string.h... yes
checking for memory.h... yes
checking for strings.h... yes
checking for inttypes.h... yes
checking for stdint.h... yes
checking for unistd.h... yes
checking for stdlib.h... (cached) yes
configure: creating ./config.status
config.status: creating Makefile
config.status: error: cannot find input file: Makefile.in
$ ls
autom4te.cache  config.log     configure     fibonacci.c
calc_fib.c      config.status  configure.ac
```

Well, that was almost magic—the autoconf program exited with an error message. As shown in the previous example, running the autoconf program produces an output file named config.log that contains information about autoconf's processing of your configure.ac file. This file lists how autoconf handles each command that it encounters, identifies variables set by autoconf as the program generates the configuration file, and lists any errors that it encounters. Though the problem in the previous example is fairly obvious, there are times when examining this log file can help you diagnose and correct problems in your configure.ac file.

In order to eliminate this error, you could remove the call to the `AC_CONFIG_FILES([Makefile])` macro, but this would still not allow the configure script to produce a Makefile. In this example, the missing link between Autoconf and the successful creation of a Makefile is the template file from which the Makefile should be created after Autoconf performs any necessary variable substitutions, as specified by the `AC_CONFIG_FILES` macro. Our configure.ac file specified that autoconf should create the file Makefile; because no special template filename was specified, the name of the input template file is Makefile.in, which has not been created yet.

There are two approaches to creating a Makefile template:

- Create it manually using your favorite text editor and including statements that take advantage of the several hundred variables that autoconf can replace with platform- and system-specific values.
- Use automake to generate an appropriate Makefile.in file from a simple configuration file called Makefile.am, and then add an automake initialization macro to the configure.ac file.

Entire books have been devoted to the topic of using GNU Make and creating Makefiles; minimizing the number of files to manually create and maintain is the whole point of the GNU build tools. The next section provides an overview of how to create a Makefile.am file. Like the configure.ac file and the configure script produced by the autoconf program, the Makefile.am file is a small text file that is automatically expanded into a robust (and complex) Makefile.in file by the automake program.

In addition to showing you how to create a simple Makefile.am file and use the automake program to create the Makefile.in file, the next section also explains a number of other files that the automake program needs to be able to find in order to package your application successfully. Once these files exist, you can then rerun your configure script and automagically create a Makefile that will compile your application, run any tests you define for the application, and even automatically create a compressed tar file that packages your application for distribution.

Creating Makefile.am Files and Other Files Required by automake

Whereas the configure.ac file used by the autoconf program sets values for application configuration by passing values to a number of well-known macros, the Makefile.am file used by the automake program creates a Makefile template based on specific make targets you define. These targets are defined using well-known variables called primaries in automake parlance. Table 7-3 describes the most commonly used automake primaries.

Table 7-3. *Commonly Used automake Primaries*

Primary	Description
`DATA`	Files that are installed verbatim by a Makefile. These are typically things like help files, configuration data, and so on.
`HEADERS`	Header files that are associated with an application.
`LIBRARIES`	Static libraries that must be installed with an application so they are available to an application at runtime.
`LTLIBRARIES`	Shared libraries that must be installed with an application so they are available to the application at runtime. The LT stands for Libtool, which is a GNU build package application used for building shared libraries. Using Libtool is explained in Chapter 8.
`MANS`	Online reference (man) pages.
`PROGRAMS`	Binary applications.
`SCRIPTS`	Executable command files that are associated with the application. Because these are not compiled, they cannot be considered `PROGRAMS`; because their protections must be specially set so they are executable, they cannot be considered `DATA`.
`SOURCES`	Source files required in order to compile an application and/or its libraries successfully in the first place.
`TESTS`	Test programs used to verify that an application is installed and executes correctly.
`TEXINFOS`	Online reference information for the info program that is in the GNU Texinfo documentation format (discussed in Appendix B).

In order to define the highest-level targets in the Makefile that will be generated from an automake Makefile.am file, you attach various prefixes to these primaries. The statements in a Makefile.am file are organized as standard `foo = bar` statements, where the target expression (`foo`) is formed by attaching a prefix to an appropriate primary, separated by an underscore. As an example, the simplest Makefile.am file that you can create defines the name of a single binary and then specifies the source files required to produce it. Continuing with the simple fibonacci program, this Makefile.am file would look like the following:

```
bin_PROGRAMS = fibonacci
fibonacci_SOURCES = fibonacci.c calc_fib.c
```

The `bin_` prefix for the `PROGRAMS` primary tells automake that the Makefile template should specify the binary directory for installation of the fibonacci program, as specified by the make variable `bindir`. The `fibonacci_` prefix before the `SOURCES` primary tells the automake program that these source files are associated with the fibonacci binary.

These two statements tell automake how to create a Makefile.in file from which autoconf can generate a Makefile that will successfully compile and install the sample application used in this chapter. The Makefiles produced by automake, listed in Table 7-4, actually provide instructions for performing a number of tasks related to the applications you are packaging.

Table 7-4. *Makefile Targets Produced by automake*

Target	Description
all	Creates a binary version of the application and builds any related data files and executable shell scripts. As with "regular" Makefiles, this is the default target if you simply execute the `make` command with no arguments against a Makefile produced by autoconf and automake.
check	Executes any tests associated with the application.
clean	Removes all binaries and intermediate object files produced during any previous execution of the Makefile.
dist	Creates a redistributable tar archive (tarball) for your application.
distcheck	Verifies that the distribution produced by `make dist` is complete and correct.
distclean	Returns the application directory to the exact state it would be in if you had freshly extracted the contents of the application archive.
install	Installs the application and any related data files and executable shell scripts.
uninstall	Uninstalls the application and any related data files and executable shell scripts from the application binary, library, and data directories.

The automake program's notion of a distribution imposes additional requirements on successfully generating a template Makefile for an application. Unlike autoconf, which only requires a single input file (configure.ac), a successful run of the automake program expects to find various auxiliary files in the directory where you are packaging your application. These files are all text files, with the names and content listed in Table 7-5.

Table 7-5. *Auxiliary Files*

File	Description
AUTHORS	A file that provides information about the authors of the program you are packaging. It typically includes the names and e-mail addresses of the primary authors, as well as information about any significant contributors to the application.
ChangeLog	A text file containing the history of changes to the application. This file lists the latest changes at the top of the file so anyone installing the application package can easily identify everything that has changed in the current distribution. The format of this file is specified in the GNU Coding Standards (`http://www.gnu.org/prep/standards.html`), though this is not enforced programmatically. This file contains a description of every change to the application source code, regardless of whether it is visible to the user.
COPYING	The GNU Public License, a copy of which is installed with the automake application.
depcomp	An auxiliary shell script that is used to compile programs and generate dependency information.
INSTALL	A template set of generic installation applications for applications that are packaged with autoconf and automake. You can customize this file to reflect any installation issues specific to your application.
install-sh	An auxiliary shell script that is used to install all of the executables, libraries, data files, and shell scripts associated with your application.
mkinstalldirs	An auxiliary shell script that is used to create any directories or directory hierarchies required by an application.
missing	An auxiliary shell script that automatically substitutes itself for any optional applications that may have been used when creating your application, but are not actually required for successfully compiling or installing your application. For example, if your distribution includes a `.c` file generated from a `.y` file by yacc or Bison, but the `.y` file has not actually been modified, you do not really need yacc or Bison on the system where you are installing the package. The missing script can stand in for programs, including aclocal, autoconf, autoheader, automake, bison, flex, help2man, lex, makeinfo, tar (by looking for alternate names for the tar program), and yacc.
NEWS	A record of user-visible changes to the application. This differs from the ChangeLog in that this file describes only user-visible changes. It does not need to describe bug fixes, algorithmic changes, and so on.
README	A file that provides initial information for new users of the application. By convention, this is the first file that someone will examine when building or installing an updated version of an application.

The automake program automatically creates the install-sh, mkinstalldirs, missing, COPYING, INSTALL, and depcomp files by copying them from templates provided as part of the automake distribution. In order to create these files, you must execute automake at least once with the `--add-missing` command-line option, as explained in the next section. You can then create the NEWS, README, AUTHORS, and ChangeLog files using a text editor or by using the touch program to create empty placeholders with the correct names.

Before we show you an example of running the automake program and creating these files in the next section, take a look at an overview of the process of using automake itself:

1. Create the Makefile.am file for your application.
2. Execute the automake program with the `--add-missing` command-line option to copy any required files missing from your distribution but for which templates are provided as part of your automake installation. This step also identifies any files you must create manually.
3. Create the files identified in the previous step.
4. Modify your application's configure.ac file to include the `AM_INIT_AUTOMAKE(program, version)` macro, customized for your application. This macro invocation should follow the `AC_INIT` macro in your configure.ac file.
5. Because the `AM_INIT_AUTOMAKE` macro is an automake macro rather than a standard autoconf macro, you will need to run the aclocal script in order to create a local file of m4 macro definitions (such as the definition of the `AM_INIT_AUTOMAKE` macro) that will be used by Autoconf.
6. Rerun the automake program again to generate a complete Makefile.in file from your Makefile.am file.

At this point, all of the files necessary for successfully using Autoconf and Automake to generate a valid configure script for your application, which will in turn generate a valid Makefile, are present.

■ **Note** United States national health regulations require that we explicitly state that attempting to read or manually modify Makefiles generated by Autoconf and Automake can be hazardous to your sanity and downright counterproductive. If you find that the automatically generated Makefile does not work correctly on your system, you should either pass options to the configure script to induce the generation of a correct Makefile (as explained in the section "Running Configure Scripts" later in this chapter), or correct the problem in the Makefile.am file, rerun automake, and then send a problem report to the maintainer of your application. If you actually find a bug in automake, you should report it to the appropriate mailing list shown in the section "Obtaining and Installing Automake."

Running Autoconf and Automake

As discussed in the previous two sections, building an application using the GNU build tools, (autoconf, automake, and libtool) requires that you first create the input files that Autoconf and Automake use as the basis of the configure script and Makefile files that they respectively create. The previous sections summarized how to create the input files required by Autoconf and Automake, but did not provide a complete "cradle-to-Makefile" example explaining the sequence in which various input files should be created and various commands executed. This section summarizes the entire process of using Autoconf and Automake to configure and build an application.

The basic sequence of using Autoconf, Automake, and the configure script to configure an application is the following:

1. Create an initial application configuration file (configure.ac), either manually or by using the autoscan script to generate a prototype that you can subsequently customize for your application.
2. Create an initial build configuration file (Makefile.am) and add the automake initialization macro statement to your configure.ac file.
3. (Optional) If your application uses a significant number of system header files, use the autoheader script to identify any system-dependent values you need to define.

■ **Note** The autoheader script only executes correctly if an `AC_CONFIG_HEADER` statement is present in your configure.ac file. If no such statement is present, the autoheader program exits with an error message.

---

4. (Optional) If your application contains any conditional compilation flags (`#ifdef`, `#ifndef`, and so on), use the ifnames script to identify these flags so you can create appropriate compilation targets or compilation flag settings in your Makefile.am file.
5. Run the aclocal script to produce the aclocal.m4 file. This file contains a local version of the autoconf macros used in the autoconf and automake configuration files to reduce the amount of work that the automake program needs to do when generating the Makefile.in file from the Makefile.am build description file.
6. Run the automake program to generate the Makefile.in script from your Makefile.am build description. This requires creating some auxiliary files that are part of a "proper" GNU application, but this is easy enough to do without significant effort.
7. Run the autoconf program to generate a configure script.
8. Execute the configure script to generate automatically and execute a shell script called config.status, which in turn creates a Makefile that will enable you to compile your application, run tests, package your application for distribution, and so on.

For trivial applications, this may seem a bit much, but it is really an investment in the future of your application. For complex applications, the hassle inherent in manually creating Makefiles and maintaining and tweaking them for every possible platform and system configuration makes the GNU build environment and tools such as Autoconf and Automake an impressive win. In general, applications rarely get simpler: using Autoconf and Automake from the beginning of any application development project can reduce the configuration, maintenance, and distribution effort required as your application becomes more complex, more popular, or (hopefully) both. The remainder of this section shows an example that walks through the entire process.

In the section "Creating configure.ac Files," the autoscan program generated a template configure.ac file for a simple application that looked like the following (with comments and blank lines removed):

```
AC_PREREQ(2.59)
AC_INIT(Fibonacci, 1.0, vonhagen@vonhagen.org)
AC_CONFIG_SRCDIR([fibonacci.c])
AC_CONFIG_HEADER([config.h])
AC_PROG_CC
AC_HEADER_STDC
AC_CHECK_HEADERS([stdlib.h])
AC_OUTPUT
```

As shown at the end of that section and explained in the section "Creating Makefile.am Files and Other Files Required by automake," the only major things missing from this file are a line telling autoconf to create a Makefile in the first place and an initialization statement for the macros used by automake. After removing the config.h line and adding an automake initialization statement and a line to create the Makefile to the sample configure.ac file, that file looks like the following:

```
AC_PREREQ(2.59)
AC_INIT(Fibonacci, 2.0, vonhagen@vonhagen.org)
AM_INIT_AUTOMAKE(fibonacci, 2.0)
AC_CONFIG_SRCDIR([fibonacci.c])
AC_PROG_CC
AC_HEADER_STDC
AC_CHECK_HEADERS([stdlib.h])
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Next, you must create a basic configuration file for the automake application. A minimal automake configuration file (automake.am) for this application looks like the following:

```
bin_PROGRAMS = fibonacci
fibonacci_SOURCES = fibonacci.c calc_fib.c
```

At this point, the working directory looks like this:

```
$ ls
```

```
calc_fib.c  configure.ac  fibonacci.c  Makefile.am
```

Once these files have been created, you run the `aclocal` command, which collects information about the location and definition of all autoconf macros and stores it in the file aclocal.m4, as in the following example:

```
$ aclocal
$ ls
```

```
aclocal.m4  calc_fib.c  configure.ac  fibonacci.c  Makefile.am
```

■ **Note** The aclocal script only creates the aclocal.m4 file if your configure.ac file contains `AM_INIT_AUTOMAKE`. Otherwise, it succeeds in doing nothing or fails silently, depending on your perspective. If you execute the `aclocal` command but do not see the file aclocal.m4, check your configure.ac file to make sure you have correctly added the `AM_INIT_AUTOMAKE` macro declaration.

Next, you must run the automake program, telling it to create any auxiliary files it needs by copying in templates from your automake installation:

```
$ automake --add-missing
```

```
configure.ac: installing './install-sh'
configure.ac: installing './mkinstalldirs'
configure.ac: installing './missing'
Makefile.am: installing './COPYING'
Makefile.am: installing './INSTALL'
Makefile.am: required file './NEWS' not found
Makefile.am: required file './README' not found
Makefile.am: required file './AUTHORS' not found
Makefile.am: required file './ChangeLog' not found
Makefile.am: installing './depcomp'
```

At this point, your working directory looks like the following:

```
$ ls
```

```
aclocal.m4      configure.ac  fibonacci.c  Makefile.am  mkinstalldirs
autom4te.cache  COPYING       INSTALL      missing
calc_fib.c      depcomp       install-sh
```

Now, you must create the other files required by the GNU build process that could not be created by copying from templates. As explained in the section “Creating Makefile.am Files and

Other Files Required by automake," these are all files that actually provide information about the history of the program and its authors, and generic information about the program that users should know before attempting to build or install it—AUTHORS, ChangeLog, NEWS, and README. The following example simply creates these files using the `touch` command, but in actual use, you should create these files with a text editor to enter the correct content:

```
$ touch AUTHORS ChangeLog NEWS README
$ ls
```

```
aclocal.m4      ChangeLog     fibonacci.c  missing
AUTHORS         configure.ac  INSTALL      mkinstalldirs
autom4te.cache  COPYING       install-sh   NEWS
calc_fib.c      depcomp       Makefile.am  README
```

Now, run the automake command to create the Makefile.in file:

```
$ automake
```

At this point, you are ready to run the autoconf program to create a configure script to generate a Makefile:

```
$ autoconf
$ ls
```

```
aclocal.m4      ChangeLog     depcomp      Makefile.am    NEWS
AUTHORS         configure     fibonacci.c  Makefile.in    README
autom4te.cache  configure.ac  INSTALL      missing
calc_fib.c      COPYING       install-sh   mkinstalldirs
```

Once you have created a configure script with no errors, the rest of the process is simple. To configure your application for your hardware and operating system, simply execute the configure script. The configure script creates a config.status script that takes all of your platform-specific information into account, and then executes that script to create a Makefile, as in the following example:

```
$ ./configure
```

```
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking for gcc... gcc
checking for C compiler default output... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ANSI C... none needed
checking for style of include used by make... GNU
checking dependency style of gcc... gcc3
checking how to run the C preprocessor... gcc -E
checking for egrep... grep -E
checking for ANSI C header files... yes
checking for sys/types.h... yes
```

```
checking for sys/stat.h... yes
checking for stdlib.h... yes
checking for string.h... yes
checking for memory.h... yes
checking for strings.h... yes
checking for inttypes.h... yes
checking for stdint.h... yes
checking for unistd.h... yes
checking for stdlib.h... (cached) yes
configure: creating ./config.status
config.status: creating Makefile
config.status: executing depfiles commands
```

Listing the contents of your working directory shows the following files:

```
$ ls
```

```
aclocal.m4          config.log          depcomp       Makefile.am       README
AUTHORS             config.status      fibonacci.c    Makefile.in
autom4te.cache      configure           INSTALL       missing
calc_fib.c          configure.ac        install-sh    mkinstalldirs
ChangeLog           COPYING             Makefile      NEWS
```

At this point, you can use the generated Makefile to build the sample application, as in the following example:

```
$ make
```

```
if gcc -DPACKAGE_NAME=\"Fibonacci\" -DPACKAGE_TARNAME=\"fibonacci\" \
  -DPACKAGE_VERSION=\"1.0\" -DPACKAGE_STRING=\"Fibonacci\ 1.0\" \
  -DPACKAGE_BUGREPORT=\"vonhagen@vonhagen.org\" \
  -DPACKAGE=\"fibonacci\" -DVERSION=\"1.0\" -DSTDC_HEADERS=1 \
  -DHAVE_SYS_TYPES_H=1 -DHAVE_SYS_STAT_H=1 -DHAVE_STDLIB_H=1 \
  -DHAVE_STRING_H=1 -DHAVE_MEMORY_H=1 -DHAVE_STRINGS_H=1 \
  -DHAVE_INTTYPES_H=1 -DHAVE_STDINT_H=1 -DHAVE_UNISTD_H=1 \
  -DHAVE_STDLIB_H=1  -I. -I.     -g -O2 -MT fibonacci.o -MD \
  -MP -MF ".deps/fibonacci.Tpo" \
  -c -o fibonacci.o 'test -f 'fibonacci.c' || echo './''fibonacci.c; \
then mv ".deps/fibonacci.Tpo" ".deps/fibonacci.Po"; \
else rm -f ".deps/fibonacci.Tpo"; exit 1; \
fi
if gcc -DPACKAGE_NAME=\"Fibonacci\" -DPACKAGE_TARNAME=\"fibonacci\" \
  -DPACKAGE_VERSION=\"1.0\" -DPACKAGE_STRING=\"Fibonacci\ 1.0\" \
  -DPACKAGE_BUGREPORT=\"vonhagen@vonhagen.org\" \
  -DPACKAGE=\"fibonacci\" -DVERSION=\"1.0\" -DSTDC_HEADERS=1 \
  -DHAVE_SYS_TYPES_H=1 -DHAVE_SYS_STAT_H=1 -DHAVE_STDLIB_H=1 \
  -DHAVE_STRING_H=1 -DHAVE_MEMORY_H=1 -DHAVE_STRINGS_H=1 \
  -DHAVE_INTTYPES_H=1 -DHAVE_STDINT_H=1 -DHAVE_UNISTD_H=1 \
  -DHAVE_STDLIB_H=1  -I. -I.     -g -O2 -MT calc_fib.o -MD \
  -MP -MF ".deps/calc_fib.Tpo" \
  -c -o calc_fib.o 'test -f 'calc_fib.c' || echo './''calc_fib.c; \
then mv ".deps/calc_fib.Tpo" ".deps/calc_fib.Po"; \
else rm -f ".deps/calc_fib.Tpo"; exit 1; \
fi
gcc  -g -O2   -o fibonacci  fibonacci.o calc_fib.o
```

```
$ ls
```

```
aclocal.m4          config.log          fibonacci       Makefile.am
AUTHORS             config.status       fibonacci.c     Makefile.in
autom4te.cache      configure           fibonacci.o     missing
calc_fib.c          configure.ac        INSTALL         mkinstalldirs
calc_fib.o          COPYING             install-sh      NEWS
ChangeLog           depcomp             Makefile        README
```

You can then execute your sample application:

```
$ ./fibonacci
```

```
Usage: fibonacci num-of-sequence-values-to-print
```

```
$ ./fibonacci  11
```

```
0 1 1 2 3 5 8 13 21 34 55
```

That is all there is to it! As explained in the section "Creating Makefile.am Files and Other Files Required by automake," the generated Makefile contains default commands for installing your program and even for running tests if you have defined some using the `TESTS` and `CHECK` primaries.

The earlier section "Installing and Configuring autoconf and automake" provided information about downloading the latest versions of Autoconf and Automake and checking the info files for the appropriate application or printing their reference manuals. For a user-oriented discussion of the GNU build tools, refer to the book *GNU AUTOCONF, AUTOMAKE, and LIBTOOL*, Gary V. Vaughan et al. (New Riders, 2000. ISBN: 1-57870-190-2). Though this book discusses substantially older versions of the GNU build tools and is therefore somewhat dated, it is an excellent and friendly book that provides detailed insights into using autoconf, automake, and libtool. (Using libtool is discussed in the next chapter of this book.)

This section explained how to create the input and auxiliary files required by Autoconf and Automake and how to execute these applications and the scripts they produce in order to compile and install your application. You may have noticed that nothing in the Autoconf or Automake input files specifies details such as the location of the resulting binaries and other application files. These values are set by passing them to the configure script produced by Autoconf, which then generates a Makefile that encapsulates this information. The next section discusses the arguments you can pass to the configure script to customize the Makefile that it creates.

Running Configure Scripts

As mentioned at the end of the previous section, the configure script generated by autoconf produces a Makefile that conforms to the instructions you provided in the Makefile.am file, but uses various defaults. For example, the default installed location of any application for any Autoconf/Automake application is in subdirectories of `/usr/local`. This is useful for user applications and for testing new versions of system applications, but at some point, you will want to configure and build a "standard" Linux application from source code and install it into some system directory. The Autoconf and Automake applications themselves are good examples of applications you may want to think about upgrading or simply replacing, as discussed in the section "Deciding Whether to Upgrade or Replace

autoconf and automake." By default, new versions of these applications that you download and build are installed in `/usr/local/bin`. Eventually, you will want to purge your system of older versions by overwriting the versions in `/usr/bin` (their default location on most Linux distributions), and then deleting your "test" versions in `/usr/local/bin`.

The configure script has a number of command-line options that enable you to customize the behavior of the Makefile that it produces. Some of these command-line options are obscure, while others are quite useful. Table 7-6 shows the most commonly used configure options and the effect that they have on the Makefile generated by the configure script.

Table 7-6. *Summary of Configure Script Options*

Option	Description
`-h`, `--help`	Display help and exit.
`-V`, `--version`	Display version information and exit.
`-n`, `--no-create`	Show you what the configure script would do, similar to `make -n`, but do not actually create output files.
`--prefix=PREFIX`	Installs files in appropriate subdirectories of `PREFIX`.

The most commonly used option with the configure script is the `--prefix` option. The default value for `PREFIX` is `/usr/local`. When you execute the configure script without the `--prefix` option, `make install` will install all of the executable files for your application in `/usr/local/bin`, all of the libraries in `/usr/local/lib`, and so on. You can specify an installation prefix other than `/usr/local` by using the `--prefix` option—for instance, use `--prefix=$HOME` to install the application in the appropriate subdirectories of your home directory, or `--prefix=/usr` to install the application in the appropriate subdirectories of `/usr`, such as `/usr/bin`, `/usr/lib`, `/usr/man`, and so on.

The configure scripts created by the Autoconf application provide a number of options that give you detailed control over various aspects of the Makefiles that are created by these scripts. For complete information about the options provided by any configure script, execute the command `./configure --help`.

CHAPTER 8

Using Libtool

This chapter discusses Libtool (the GNU Library Tool), an open source application that simplifies creating and maintaining precompiled collections of functions (for languages such as C) or classes and associated methods (in object-oriented languages such as C++). These functions and classes, generally referred to as *libraries*, can subsequently be used by and linked to applications. Throughout this chapter, the term *code libraries* is used as a general, language-independent term for a precompiled collection of functions or classes that is external to an application. These code libraries can be linked to an application during final compilation, enabling it to execute the functions or methods in the library.

This chapter begins by explaining the basic principles of libraries, the different types of libraries that are used in application development, and the way each is used by applications. It then introduces Libtool and explains how to download, build, and use Libtool when building applications. The last sections explain potential problems you may encounter when using Libtool, outline how to correct or work around those problems, and summarize additional sources of information about Libtool and its use.

Introduction to Libraries

A library is a collection of precompiled code that can be used by any application that is linked to it. Linking usually occurs during the final stage of compiling an application. Source code modules that use functions located in libraries are typically compiled with compiler options such as GCC's `-c` option, which tells the compiler to compile the source module but not to attempt to resolve references to functions that are not located in that source module. These references are resolved during the final stage of compilation when an executable version of the application is actually produced. How these references are resolved depends on the type of library that the executable is being linked with.

Three basic types of libraries are available on most modern computer systems: static, shared, and dynamically loaded. The next sections define each of these and discuss their advantages and disadvantages.

Static Libraries

Static libraries are the oldest and simplest form of code libraries. Each application that is linked with a static library essentially includes a copy of the library in the final executable version of the application. Using static libraries is fairly simple because references to function calls in the code library can be resolved during compilation. When linking a static library to an application, the address of any function in the library can be determined and preresolved in the resulting executable. Static libraries typically use the extension `.a` (for archive) or `.sa` (for static archive).

Aside from their relative simplicity during compilation, a significant advantage of using static libraries is that applications that are statically linked with the libraries they use are completely self-sufficient. The resulting executable can be transferred to any compatible computer system and will execute correctly there because it has no external dependencies. Applications that are statically linked to the libraries they use can be executed more quickly than applications that use shared or dynamically loaded libraries, because statically linked applications require no runtime or startup overhead for resolving function calls or class references. Similarly, statically linked applications require somewhat less memory than shared or dynamically loaded applications, because they require a smaller runtime environment with less overhead than is required to search for and call functions in a shared or dynamically loaded library.

While static libraries are easy to use and have some advantages for portability, they also have some distinct disadvantages, the most significant of which are the following:

- *Increased application size*: Bundling a copy of a static library into each executable that uses it substantially increases the size of each resulting executable. The portions of the library used in the application must be bundled into each executable because it is not possible to extract specific functions from the library.
- *Recompiling applications after updates*: Any bug fixes or enhancements made to libraries that have been statically linked into applications are not available to those applications unless the applications are recompiled with the new version of the library.

To address these problems, most Unix and Unix-like systems support shared and dynamically loaded libraries, which are discussed in the next sections.

Shared Libraries

First introduced by Sun Microsystems in the early 1990s, shared libraries are centralized code libraries that an application loads and links to at runtime, rather than at compile time. Programs that use shared libraries only contain references to library routines. These references are resolved by the runtime link editor when the programs are executed. Shared libraries typically have the file extension `.so` (for shared object), though they may use other extensions, such as the `.dylib` extension (for dynamic shared library) used on Mac OS X systems. Mac OS X is something of an anomaly in terms of how it names and references shared libraries. The remainder of this section primarily focuses on systems that use "classic" shared-library implementations such as Linux and Solaris.

Shared libraries provide a number of advantages over static libraries, most notably the following:

Simplified application maintenance: Shared libraries can be updated at any time, and they use a version numbering scheme so applications can load the "right" version of a shared library. Applications compiled against a shared library can automatically take advantage of bug fixes or enhanced functionality provided in newer versions of those libraries. Applications compiled against specific or previous-generation shared libraries can continue to use those libraries without disruption. Shared library version numbering is explained later in this section.

Reduced system disk space requirements: Applications that use shared libraries are smaller than applications that are statically linked with libraries because the library is not bundled into the application. This is especially significant in graphical applications, such as X Window System applications, that use functions contained in a relatively large number of graphics libraries. Compiling these types of applications with static libraries would increase their size three or four times. Similarly, running a graphical desktop environment, such as GNOME, KDE, or X Window manager and some number of X Window System applications that use shared libraries, can significantly reduce overall system memory consumption, since all of these applications share access to a single copy of each shared library rather than including their own copy.

Reduced system memory requirements: Applications that access shared libraries still need to load the shared libraries into memory when these applications are executed. However, shared libraries need only be loaded into system memory once in order to be used by any application that references the library. Because multiple applications can simultaneously share access to a single copy of a centralized shared library, overall system memory consumption is reduced when more than one application that references a shared library is being executed.

Using shared libraries also has some negative aspects. First, using shared libraries makes it more complex to copy single executables between compatible computer systems, because it increases the external requirement for the application. Each computer system must provide the shared libraries, as well as appropriate versions of these shared libraries. For example, consider the X Window System xterm application on a modern Linux system. The Linux `ldd` (list dynamic dependencies) command shows that the xterm application requires the following shared libraries:

```
$ ldd 'which xterm'
```

```
        libXft.so.1 => /usr/X11R6/lib/libXft.so.1 (0x4002f000)
        libfreetype.so.6 => /usr/lib/libfreetype.so.6 (0x4003e000)
        libXrender.so.1 => /usr/X11R6/lib/libXrender.so.1 (0x40087000)
        libXaw.so.7 => /usr/X11R6/lib/libXaw.so.7 (0x4008c000)
        libXmu.so.6 => /usr/X11R6/lib/libXmu.so.6 (0x400e6000)
        libXt.so.6 => /usr/X11R6/lib/libXt.so.6 (0x400fc000)
        libSM.so.6 => /usr/X11R6/lib/libSM.so.6 (0x4014e000)
        libICE.so.6 => /usr/X11R6/lib/libICE.so.6 (0x40158000)
        libXpm.so.4 => /usr/X11R6/lib/libXpm.so.4 (0x4016f000)
        libXext.so.6 => /usr/X11R6/lib/libXext.so.6 (0x4017e000)
        libX11.so.6 => /usr/X11R6/lib/libX11.so.6 (0x4018c000)
        libncurses.so.5 => /usr/lib/libncurses.so.5 (0x4026a000)
        libutempter.so.0 => /usr/lib/libutempter.so.0 (0x402a9000)
        libc.so.6 => /lib/i686/libc.so.6 (0x42000000)
        libfontconfig.so.1 => /usr/lib/libfontconfig.so.1 (0x402ac000)
        libdl.so.2 => /lib/libdl.so.2 (0x402d1000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
        libexpat.so.0 => /usr/lib/libexpat.so.0 (0x402d4000)
```

The second downside of using shared libraries is that their use imposes a slight performance penalty when any application using the shared library is executed, because references to shared library routines must be resolved by finding the libraries containing those routines at runtime.

At runtime, resolving references to shared libraries is done by a shared library loader. Systems that use shared libraries typically maintain a cache that identifies their locations in order to minimize time when an application that uses shared libraries is executed. For example, Linux systems use the shared library loader /lib/ld.so and maintain on your system a cache of the locations of shared libraries in the file /etc/ld.so.cache. The cache file is created and maintained using the ldconfig application, based on the contents of the file /etc/ld.so.conf, which contains a list of directories to search for shared libraries.

In order for systems and applications that use shared libraries to support multiple versions of those libraries and to distinguish between them, most shared library implementations use a common set of naming conventions for those libraries. The actual name of a shared library is composed of the prefix `lib`, the name of the library, and the basic shared library file extension `.so`, followed by a period and the true version number of the shared library. The version number is either two or three digits. On Solaris systems, the version number is composed of a major and a minor number separated by a period, such as `libc.so.2.9`. Linux systems use the same naming scheme, but also add a release number to the end, such as `libxmms.so.1.2.1`.

Shared libraries also have a special name called the soname. The soname is composed of the prefix `lib`, the name of the library, and the basic shared library file extension `.so`, followed by a period and the major version number of the shared library. The soname is typically a symbolic link to the appropriate shared library file. Continuing with the examples in the previous paragraph, the soname of the Solaris `libc.so.2.9` library would be `libc.so.2`, whereas the soname of the Linux `libxmms.so.1.2.1` library would be `libxmms.so.1`.

The final name used to refer to shared libraries is often called the linker name. This name, which a compiler uses when referring to a library, is simply the soname without any version number. Continuing with the examples in the preceding paragraphs, the linker name of the Solaris `libc.so.2.9` library would be `libc.so`, whereas the linker name of the Linux `libxmms.so.1.2.1` library would be `libxmms.so`.

The version numbering scheme used in shared library names makes it possible to maintain several compatible and incompatible versions of a single shared library on the same system. The major, minor, and release version numbers of a shared library are updated based on the type of changes made between different versions of the shared library. When a new version of a shared library is released in order to fix bugs in a previous version, only the release number is incremented. When new functions are added to a shared library but existing functions in the library still provide the same interfaces, the minor version number is incremented and the release number is reset. When interfaces in functions in a shared library change in such a way that existing applications cannot call those functions, the major number is updated and the minor and release numbers are reset. Applications that depend on specific interfaces can still load the shared library with the correct major and minor number at runtime, while applications compiled against a new version of the same shared library (featuring different parameters or a different parameter sequence) to existing functions can link against the version of the shared library with the new major number.

Dynamically Loaded Libraries

The final type of libraries in common use are dynamically loaded (DL) libraries. Dynamically loaded libraries are code libraries that an application can load and reference at any time while it is running, rather than arbitrarily loading them all at runtime. Unlike static and shared libraries, which have different formats and are automatically loaded in different ways, dynamically loaded libraries differ from other types of libraries in terms of how an application uses them, rather than how the runtime environment or compiler uses them. Both static and shared libraries can be used as dynamically loaded libraries.

The most common use for DL libraries is in applications that use plug-ins or modules. A good example of a common use of dynamically loaded libraries is the Pluggable Authentication Modules (PAM) employed by the authentication mechanism on many Linux, FreeBSD, Solaris, and HP-UX systems. The primary advantage of DL libraries is that they enable you to defer the overhead of loading specific functionality unless it is actually required. DL libraries also make it easy to separately develop, maintain, and distribute plug-ins independent of the executable that actually invokes and loads them. They are also useful in CORBA and Com environments where you may not actually know which library to load until runtime.

Because they are loaded "as required," an application that wishes to use DL libraries does not need any prior knowledge of the libraries other than their names or, more commonly, the locations to search for them. Most applications that use dynamically loaded modules use some sort of indirection to determine the DL libraries that are available—hardwiring a list of available plug-ins into an application largely defeats the flexibility they provide. Instead, applications that use DL libraries typically use a well-known list of directories to search for DL libraries, or read a list of relevant directories from an environment variable or configuration file.

Applications that use DL libraries employ a standard API for opening DL libraries, examining their contents, handling errors, and so on. In C language applications on Linux systems, this API is defined and made available by including the system header file <dlfcn.h>. The actual functions in the API (unfortunately) differ across platforms—for example, Linux, Solaris, and FreeBSD systems use the `dlopen()` function to open libraries, HP-UX systems use `shl_load()`, and Windows systems use the radically different (of course) LoadLibrary interface for their dynamic link libraries (DLLs). The GNU Glibc library provides a more generic interface to DL libraries that is intended to hide most platform-specific differences in DL library implementations. The Libtool library tool, the subject of this chapter, also provides a general interface for DL libraries through its `libltdl.so` library.

What Is Libtool?

Even creating a simple static library and using it in your applications requires special compilation and linking options. The various types of libraries discussed in the previous section, the differences in using them in applications, and the differences in how or if applications need to be able to resolve symbols in those libraries each brings its own complexity and caveats to developing applications that use libraries. If you are writing an application that uses libraries on multiple platforms, multiplying these considerations across target systems and library implementations is the stuff of nightmares.

Libtool consists of a shell script and auxiliary libraries that are designed to encapsulate platform- and library-format-specific differences. Libtool does this by providing a consistent user interface that is supported on a wide variety of platforms. Libtool is most commonly invoked by the configure scripts and Makefiles produced by Autoconf and Automake. However, like all good GNU utilities, Libtool does not depend on Autoconf or Automake, and vice versa. Libtool is just another tool that configure scripts and Makefiles can use if it is available on the system where you are building your application. Manually created Makefiles can invoke Libtool just as easily as can those created by Automake.

Libtool introduces some new conventions in library extensions that are designed to make differences between the types of libraries supported on different types of systems. Though these are really internal to Libtool, it is useful to discuss them up front to minimize confusion if you are wondering what is going on when you see new types of files being created during the build process. Most notably, the majority of C language compilers use the extension convention `.o` for intermediate object files produced during the compilation process. To simplify things internally across systems that may or may not support shared libraries, Libtool creates files with the extensions `.la`, which are files that contain platform-specific information about your libraries, and `.lo` (library object), which are optional files produced during the build process. When generated by Libtool on systems that do not support shared libraries, these `.lo` files contain time stamps rather than actual object code. When actually building libraries, Libtool uses the information about your system's library support to determine how to (and if you should) use the library object files, and employs the information in the `.la` files to identify the characteristics of your system's support for libraries and the actual libraries that should be used.

Aside from masking platform-specific differences in library construction and use, the most interesting problem that Libtool solves is the problem of linking applications against shared libraries that are being developed at the same time, and are therefore not preinstalled in one of the locations your system searches for shared libraries. Though you could manually resolve this problem by setting an environment variable such as `LD_LIBRARY_PATH` to include the directories where the shared libraries you are developing are located, this would truly be a pain in complex applications that use multiple shared libraries. Libtool handles this for you automatically, which is well worth the cost of admission all by itself (especially when that cost is zero). Libtool is an elegant but easy-to-use solution to a variety of complex and/or irritating problems.

Downloading and Installing Libtool

In order to provide a consistent interface across many different types of systems, each of which may use and support different libraries, you must explicitly build Libtool on each platform on which you plan to use it. Even though it is a shell script, the script contains a number of platform-specific variable settings that are required in order for Libtool to identify and take advantage of the capabilities of your system.

Most systems that use GCC also provide auxiliary tools such as Libtool, Automake, and Autoconf. However, if you are building GCC for your system or simply installing the latest and greatest version, you will also want to obtain, build, and install the latest version of Libtool. A link to a downloadable archive of the source code for the latest version of Libtool is available from the Libtool home page at `http://www.gnu.org/software/libtool/libtool.html`. However, I recommend you use Libtool 1.5.18, because installing a newer version may cause conflicts with portions of any version of Libtool already installed on your system. Newer versions are fine if you are absolutely sure that Libtool is not currently installed on your system. Though not used in this section, Libtool 1.9f was the newest version (a developer release) available at the time of this writing. You can download Libtool 1.5.18 from the Libtool page at any GNU mirror, such as `ftp://mirrors.kernel.org/gnu/libtool`.

The remainder of this section will use an archive file named `libtool-1.5.18.tar.gz` and source code directory named `libtool-1.5.18` as examples—when following the instructions in this section on your system, you should substitute the name of the archive file you downloaded and the name of the source code directory you extracted from it.

Installing Libtool

Once you have downloaded the archive, you can extract its contents using a command such as the following:

```
# tar zxf libtool-1.5.18.tar.gz
```

This will create a directory with the basename of the tar file that contains the Libtool source code. In this example, the name of the extracted directory would be `libtool-1.5.18`.

Make the extracted directory your working directory and execute the configure script provided as part of the Libtool source distribution.

```
# cd libtool-1.5.18
# ./configure
  [standard configure output deleted]
```

UPDATING OR OVERWRITING LIBTOOL

If you want to install the new version of Libtool in the appropriate subdirectories of a directory other than the default directory `/usr/local`, you can use the configure script's `--prefix` option to specify the new directory hierarchy. For example, if you want to install Libtool in the subdirectories of `/usr`, you would execute the command `./configure --prefix=/usr`. You may want to do this if you are updating a system, such as a default Red Hat Linux distribution, on which Libtool is already installed. If you are using a package-based computer system, you may want to use your system's package management application to delete the package that provides Libtool so your system does not think another version is installed. For example, on systems that use the Red Hat Package Manager (RPM), you can identify and remove the package that provides Libtool by using commands such as the following:

```
# rpm -qf 'which libtool'
```

```
libtool-1.5.18-12
```

```
# rpm -e libtool-1.5.18-12
```

There is really no need to remove previous versions of Libtool, except that it avoids potential confusion should you encounter a problem and need to provide someone with specific information about the version of Libtool and associated files you are using.

Next, use the `make` command to build Libtool (output included due to its brevity):

```
# make
```

```
Making all in .
make[1]: Entering directory `/home/wvh/src/libtool-1.5.18'
CONFIG_FILES=libtoolize CONFIG_HEADERS= /bin/sh ./config.status
config.status: creating libtoolize
config.status: executing depfiles commands
chmod +x libtoolize
make[1]: Leaving directory `/home/wvh/src/libtool-1.5.18'
Making all in libltdl
make[1]: Entering directory `/home/wvh/src/libtool-1.5.18/libltdl'
make  all-am
make[2]: Entering directory `/home/wvh/src/libtool-1.5.18/libltdl'
/bin/sh ./libtool --tag=CC --mode=compile gcc -DHAVE_CONFIG_H -I. ➥
    -I. -I.     -g -O2 -c -o ltdl.lo ltdl.c
mkdir .libs
 gcc -DHAVE_CONFIG_H -I. -I. -I. -g -O2 -c ltdl.c  -fPIC -DPIC -o .libs/ltdl.o
 gcc -DHAVE_CONFIG_H -I. -I. -I. -g -O2 -c ltdl.c -o ltdl.o >/dev/null 2>&1
/bin/sh ./libtool --tag=CC --mode=link gcc  -g -O2   -o libltdl.la ➥
    -rpath /usr/local/lib -no-undefined -version-info 4:1:1 ltdl.lo -ldl
gcc -shared  .libs/ltdl.o  -ldl  -Wl,-soname -Wl,libltdl.so.3 ➥
    -o .libs/libltdl.so.3.1.1
(cd .libs && rm -f libltdl.so.3 && ln -s libltdl.so.3.1.1 libltdl.so.3)
(cd .libs && rm -f libltdl.so && ln -s libltdl.so.3.1.1 libltdl.so)
```

```
ar cru .libs/libltdl.a  ltdl.o
ranlib .libs/libltdl.a
creating libltdl.la
(cd .libs && rm -f libltdl.la && ln -s ../libltdl.la libltdl.la)
make[2]: Leaving directory `/home/wvh/src/libtool-1.5.18/libltdl'
make[1]: Leaving directory `/home/wvh/src/libtool-1.5.18/libltdl'
Making all in doc
make[1]: Entering directory `/home/wvh/src/libtool-1.5.18/doc'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `/home/wvh/src/libtool-1.5.18/doc'
Making all in tests
make[1]: Entering directory `/home/wvh/src/libtool-1.5.18/tests'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `/home/wvh/src/libtool-1.5.18/tests'
```

You can now install this version of Libtool and its associated scripts and libraries using the `make install` command. The previous example showed the extraction and compilation process being done as root; this is not necessary to download and compile Libtool, but you will almost certainly have to be the root user in order to install it.

Congratulations! You have built and installed Libtool!

Files Installed by Libtool

The files and directories listed in this section are installed in subdirectories of `/usr/local` by default. Many Linux distributions install these files and directories as subdirectories of `/usr` instead. See the note in the previous section for information on modifying the default location where Libtool and related files and directories will be installed. This section discusses the location of files installed by a default installation of Libtool.

The files that are installed as part of Libtool are the following:

- `config.guess`: A shell script used when Libtool is employed with Autoconf in order to guess the system type. This script is installed in `/usr/local/_share/libtool` by default.
- `config.sub`: A shell script used when Libtool is employed with Autoconf, primarily containing routines to validate your system's operating system and configuration. This script is installed in `/usr/local/share/libtool` by default.
- `libltdl.a`, `libltdl.la`, `libltdl.so`, `libltdl.so.3`, `libltdl.so.3.1.0`: The actual library files and related symlinks, soname, etc., required to use Libtool's DL library interface (ltdl). These files are installed in `/usr/local/lib` by default.
- `libtool`: The primary Libtool shell script, installed by default in `/usr/local/bin`.
- `libtool.m4`: A collection of macro definitions used when Libtool is employed with Autoconf and Automake. This file is installed in `/usr/local/share/aclocal` by default.
- `libtool/libltdl`: A directory containing the files necessary to build Libtool's DL library interface (ltdl) using Autoconf and Automake. This directory contains the files `README`, `stamp-h.in`, `COPYING.LIB`, `Makefile.am`, `Makefile.in`, `acinclude.m4`, `aclocal.m4`, `config-h.in`, `configure`, `configure.in`, `ltdl.cm`, `ltdl.c`, and `ltdl.h`. This directory is installed in `/usr/local/share` by default.

- `libtoolize`: A shell script that creates the files required to use Libtool with Autoconf and Automake. This script is installed by default in `/usr/local/bin`.
- `ltdl.h`: The include file necessary to use Libtool's DL library interface (ltdl) in an application. This file is installed in `/usr/local/include` by default.
- `ltdl.m4`: A collection of macro definitions used when Libtool is employed with Autoconf and Automake, and then only when Libtool's DL library interface (ltdl) is being used. This file is installed in `/usr/local/share/aclocal` by default.
- `libtool.info`, `libtool.info-1` *through* `libtool.info-5`: The files used by the info program to display the up-to-date documentation for Libtool. Like most GNU programs, the documentation provided for Libtool in info format is more extensive and more up-to-date than that provided in its main page. These files are installed in `/usr/local/info` by default.
- `ltmain.sh`: A collection of shell functions that are used by Libtool. This file is installed in `/usr/local/share/libtool` by default.

Using Libtool

The easiest way to explain how to use Libtool is to actually provide some examples. If you are writing an application that uses libraries, we suggest that you also use Autoconf and Automake, and simply incorporate the appropriate Libtool macros in the Makefile.am file that you created for use by Automake. A large part of the amazing beauty of the GNU configuration and compilation tools is how well they work together, and how trivial it is to use them together automatically. Today's developers truly have the opportunity to stand on the shoulders of GNU giants.

The previous paragraph aside, Libtool has an interesting interface that you may want to invoke manually at times. The next section, "Using Libtool from the Command Line," explains the commands available within Libtool and how to use them manually. The section "Using Libtool with Autoconf and Automake" explains the statements you can add to your Makefile.am file in order to automatically invoke the Libtool script.

■ **Note** Though Libtool commands can be manually integrated into Makefiles, there is not much point in doing that unless you are a true Luddite or a masochist. I therefore do not explicitly discuss that here. If you are interested in doing that, you can always examine a Makefile that incorporates Libtool support that was produced by the Autotools, and reverse-engineer manual integration of Libtool into your Makefiles.

Using Libtool from the Command Line

When used from the command line, Libtool enables you to execute various development-related commands by specifying particular command-line options and/or a particular mode of operation. These modes of operation use Libtool as a wrapper to encapsulate the functionality of development and diagnostic tools such as GCC and GDB, while still taking advantage of the functionality provided by Libtool.

The next section discusses Libtool's command-line options and their meaning. After that you'll learn about the modes of operation available in Libtool and any mode-specific command-line options that are available. This section also provides specific examples of using Libtool in each mode.

Command-Line Options for Libtool

Libtool provides a number of command-line options and associated parameters that enable you to specify a mode of operation, debug its execution, display information about its configuration and associated variables, and so on. Table 8-1 shows the command-line options provided by Libtool.

Table 8-1. *Libtool's Command-Line Options*

Option	Description
`--config`	Displays information about the characteristics of the host and operating system on which Libtool was configured, and associated Libtool variable settings.
`--debug`	Activates the shell's verbose mode (`-x`) to provide tracing information. Using this option shows each line in the Libtool script as it executes, enabling you to monitor and debug its operation by seeing which branches are taken by Libtool; shows the commands and associated options that Libtool executes; and so on.
`--dry-run`	Displays the commands that would be executed by Libtool based on the other options that are specified, but does not actually execute those commands. This option is analogous to the `make` command's `-n` option. To reinforce this parallelism, the `-n` option can also be used as an equivalent to the `--dry-run` option.
`--features`	Displays information about the type of host on which Libtool was compiled, and whether that platform supports shared and/or static libraries.
`--finish`	Specifies `finish` mode by using the `--mode` option (a shortcut).
`--help`	Displays information about the options supported by Libtool and then exits.
`--mode=MODE`	Enables you to specify a mode of operation for Libtool. Modes are provided as shorthand for the commands required to perform a specified type of conceptual operation. For example, using the `--mode=compile` option invokes GCC and any related portions of Libtool. Libtool provides a number of supported modes, many of which have additional command-line options that are specific to a given mode. Libtool's modes are discussed in the next section.
`--quiet`	Suppresses the display of status messages during Libtool execution.
`--silent`	Suppresses the display of status messages during Libtool execution, identical to the `--quiet` option.
`--version`	Displays the version of Libtool that is being executed and then exits.

The next section discusses the various modes of operation that can be specified using the `--mode` command-line option, and any additional command-line options that are specific to each mode of operation. Where possible, this section uses examples from the sample fibonacci program in Chapter 7.

Command-Line Modes for Libtool Operation

As discussed in the previous section, a variety of conceptual tasks can be performed by Libtool by specifying them using Libtool's `--mode` command-line argument. Some of these modes provide additional command-line options that are specific to a given mode.

The following modes of operation and additional mode-specific command-line options are available in Libtool:

- `clean`: Removes files from the build directory. This mode requires that you specify the name of the command used to remove files, any options that you want to pass to the command, and the name(s) of the file(s) that you want to remove. If any of the files that you specify for removal are Libtool objects or libraries, all of the files associated with them are removed. For example, the following command only removes a single object file:

  ```
  $ libtool --mode=clean rm fibonacci.o
  ```

```
rm fibonacci.o
```

- The following command, where the file to be removed is a Libtool library, removes all files associated with that library:

  ```
  $ libtool --mode=clean rm libfib.la
  ```

```
rm libfib.la .libs/libfib.so.1.0.0 .libs/libfib.so.1 .libs/libfib.so \
               .libs/libfib.a .libs/libfib.la .libs/libfib.lai
```

- `compile`: Compiles a source file into a Libtool object. This mode requires that you specify the name of the compiler and any mandatory compiler options. For example, the following command compiles the main module of the sample application into a standard object file and a Libtool library object:

  ```
  $ libtool --mode=compile gcc -c fibonacci.c
  ```

```
rm -f .libs/fibonacci.lo
gcc -c fibonacci.c  -fPIC -DPIC -o .libs/fibonacci.lo
gcc -c fibonacci.c -o fibonacci.o >/dev/null 2>&1
mv -f .libs/fibonacci.lo fibonacci.lo
```

- Some additional command-line options are available when executing Libtool in `compile` mode. These are the `-o` option, which must be followed by the name of the object file whose creation you want to force; the `-prefer-pic` option, which tells Libtool to attempt to build position-independent code (PIC) objects (required for shared libraries produced by Libtool); the `-prefer-non-pic` option, which tells Libtool to try to build non-PIC library objects (required for static libraries produced by Libtool) and overrides the default settings for your platform if it supports shared libraries; and the `-static` option, which forces Libtool to produce only a standard object file (`.o` file) that can be used to produce a standard static library.
- `execute`: Automatically sets the environment variable used to identify the location of required libraries (`LD_LIBRARY_PATH`), and then attempts to execute the specified program. This will work only if the executable and all shared or static libraries that it requires have successfully been compiled, as shown in this example:

```
$ libtool --mode=execute ./fibonacci
```

```
lt-fibonacci: error while loading shared libraries: \
  libfib.so.1: cannot open shared object file: No such file or directory
```

```
$ make >/dev/null 2>&1
$ libtool --mode=execute ./fibonacci
```

```
Usage: fibonacci num-of-sequence-values-to-print
```

- In order to completely execute the specified argument, you must also specify any command-line arguments that the application itself requires. Because Libtool does a certain amount of quoting when passing arguments to an application executed in this fashion, this actually exposes a bug in the sample fibonacci application, which does not check whether the argument being passed is actually an integer. Physician, heal thyself!
- One additional command-line option is available when executing Libtool in `execute` mode. This is the `-dlopen` option, which takes one argument, the full pathname of a shared library that you want to add to your `LD_LIBRARY_PATH` environment variable prior to executing the command you specified.
- `finish`: Completes the installation of installed libraries by executing the system `ldconfig` command to add a specified directory to the system's working `LD_LIBRARY_PATH`. To permanently add a directory to the system's shared library search path, you should add the name of that directory to the configuration file for your loader. On most Linux systems, this is the file `/etc/ld.so.conf`. You must then rerun the `ldconfig` command to add the new directory to your system's library cache.
- `install`: Installs libraries or executables. This Libtool command provides a manual way of forcing the execution of specified files. This is typically more easily done using the `make install` command from the command line, but can be useful if you want to force the execution of a single component of your application into a specified directory for testing purposes. One advantage of executing this from the command line is that Libtool understands the format of its library objects files, and installs all of the files associated with a specified library, as in this example:

  ```
  # libtool --mode=install install libfib.la /usr/local/lib
  ```

```
install .libs/libfib.so.1.0.0 /usr/local/lib/libfib.so.1.0.0
(cd /usr/local/lib && rm -f libfib.so.1 \
     && ln -s libfib.so.1.0.0 libfib.so.1)
(cd /usr/local/lib && rm -f libfib.so \
     && ln -s libfib.so.1.0.0 libfib.so)
install .libs/libfib.lai /usr/local/lib/libfib.la
install .libs/libfib.a /usr/local/lib/libfib.a
ranlib /usr/local/lib/libfib.a
chmod 644 /usr/local/lib/libfib.a
PATH="$PATH:/sbin" ldconfig -n /usr/local/lib
----------------------------------------------------------------------
Libraries have been installed in:             /usr/local/lib

If you ever happen to want to link against installed libraries
in a given directory, LIBDIR, you must either use libtool, and
specify the full pathname of the library, or use the '-LLIBDIR'
```

```
flag during linking and do at least one of the following:
             - add LIBDIR to the 'LD_LIBRARY_PATH' environment variable
     during execution
   - add LIBDIR to the 'LD_RUN_PATH' environment variable
     during linking
   - use the '-Wl,--rpath -Wl,LIBDIR' linker flag
   - have your system administrator add LIBDIR to '/etc/ld.so.conf'

See any operating system documentation about shared libraries for
more information, such as the ld(1) and ld.so(8) manual pages.
----------------------------------------------------------------------
```

- `link`: Creates a library or an executable. This Libtool mode requires that you specify a complete link command. Neglecting to do so is the most common first-time error when using Libtool in this mode, as shown in the following example:

  ```
  $ libtool --mode=link fibonacci
  ```

```
libtool: link: you must specify an output file
Try 'libtool --help --mode=link' for more information.
```

- The next most common error when using this Libtool mode is neglecting to specify libraries required by the application, as in the following example:

  ```
  $ libtool --mode=link gcc fibonacci.c -o fibonacci
  ```

```
gcc fibonacci.c -o fibonacci
/tmp/ccWaF9hG.o: In function 'main':
/tmp/ccWaF9hG.o(.text+0x5b): undefined reference to 'calc_fib'
collect2: ld returned 1 exit status
Using this Libtool mode in the correct way looks something like the following:
$ libtool --mode=link gcc fibonacci.c -o fibonacci -lfib
gcc fibonacci.c -o .libs/fibonacci  \
    /home/wvh/writing/gcc/src/.libs/libfib.so -Wl,--rpath \
-Wl,/usr/local/lib
creating fibonacci
```

Libtool's link mode has more mode-specific command-line options than any other mode, as you can see by the list that follows. All of these can be specified as part of your link command:

- `-all-static`: Specifies to not do any dynamic linking. If the target of the link operation is a library, a static library will be created. If it is an executable, it will be statically linked.
- `-avoid-version`: Specifies to not add a version suffix to libraries, if possible.
- `-dlopen file`: Causes the resulting executable to try to preopen the specified object file or library and add its symbols to the linked executable if it cannot be opened at runtime.
- `-dlpreopen file`: Links the specified object file or library and adds its symbols to the linked executable.
- `-export-dynamic`: Enables symbols from the linked executable to be resolved with `dlsym(3)` rather than adding them to the symbol table for the linked executable at link time.
- `-export-symbols symbol_file`: Limits exported symbols to those specified in the file `SYMBOL_FILE`.

- `-export-symbols-regex regex`: Limits exported symbols to those that match the regular expression specified in regex.
- `-Llibdir`: Searches the specified `libdir` directory for required libraries. This can be useful if your application uses static libraries that are already installed in a nonstandard location that is not part of your library load path.
- `-lname`: Specifies that the linked executable requires symbols that are located in the installed library with a base or soname of `libname`.
- `-module`: Specifies that the link command builds a dynamically linked library that can be opened using the `dlopen(3)` function.
- `-no-fast-install`: Disables fast-install mode, which means that executables built with this flag will not need relinking in order to be executed from your development area.
- `-no-install`: Links an executable that is designed only to be executed from your development area.
- `-no-undefined`: Verifies that a library you are linking does not refer to any external, unresolved symbols.
- `-o output-file`: Specifies the name of the output file created by the linker.
- `-release release`: Specifies that the library was generated by a specific release (`release`) of your software. If you want your linked executable to be binary compatible with versions compiled using other values of `release`, you should use the `-version-info` option instead of this one.
- `-rpath libdir`: Specifies that a library you are linking will eventually be installed in the directory `libdir`.
- `-R libdir`: Adds the specified directory (`libdir`) to the runtime path of the programs and libraries that you are linking.
- `-static`: Specifies to not do any dynamic linking of libraries.
- `-version-info current[:revision[:age]]`: Enables you to specify version information for a library that you are creating. Unlike the `-release` option, libraries and executables linked with different values for the `-version` option maintain binary compatibility with each other.
- `uninstall`: Removes libraries from an installed location. This Libtool mode requires that you specify the name of the command used to uninstall these libraries (typically `/bin/rm`) and any options that you want to provide to that command, as well as the name of the library or libraries that you want to remove. For example:

```
# libtool --mode=uninstall /bin/rm /usr/local/lib/libfib.la
```

```
/bin/rm /usr/local/lib/libfib.la /usr/local/lib/libfib.so.1.0.0 \
        /usr/local/lib/libfib.so.1 /usr/local/lib/libfib.so \
        /usr/local/lib/libfib.a
```

Libtool's primary goal is to simplify and hide the complexity of building libraries, so few people actually run Libtool from the command line—it is usually integrated into projects built using Autoconf and Automake, as explained in the next section. However, you can also manually create Makefiles that explicitly call out the Libtool steps and which therefore enable you to execute each step separately during development and debugging. A sample Makefile for the fibonacci example that I've been using throughout this book might look like the following:

```
.DUMMY: all library app link install
library: libfib.c
        libtool --mode=compile gcc -c libfib.c

app: fibonacci.c
        libtool --mode=compile gcc -c fibonacci.c

link:
        libtool --mode=link gcc fibonacci.o libfib.o -o fib

install:
        cp fib /usr/local/bin
        libtool --mode=install install libfib.lo /usr/local/lib

all: library app link install
```

Using Libtool with Autoconf and Automake

Chapter 7 of this book explained how to create the configure.ac project configuration file used by Autoconf, and the Makefile.am project configuration file used by Automake. These sections of Chapter 7 used the sample application introduced in Chapter 6, consisting of the two source code modules shown in Listings 6-1 and 6-2. For convenience sake, the sample configure.ac file for this application, discussed in detail in the section "Creating configure.ac Files" in Chapter 7, is shown again in Listing 8-1. The sample Makefile.am file for this application, discussed in detail in the section "Creating Makefile.am Files and Other Files Required by Automake" in Chapter 7, is shown again here in Listing 8-3.

This section explains how to integrate Libtool into the configuration files used by Autoconf and Automake. I will demonstrate how to modify the sample fibonacci application so the calculation routine provided in the file calc_fib.c (shown in Listing 6-2) is actually compiled into a library rather than a stand-alone object module as it was in Chapter 7. This highlights an impressive advantage of the integration of Libtool and the Autoconf and Automake configuration tools—a change such as this can be done by simply modifying the configuration files used by these applications, and "the right thing" happens.

Listing 8-1 shows the initial configure.ac file for the sample application. Listing 8-2 shows this file after the addition of the `AC_PROG_LIBTOOL` macro, which is required in order to tell Autoconf that Libtool is available for use during configuration. In this example, this is the only change that needs to be made to the configure.ac file, because most of the work required to change the `calc_fib.c` routine from a stand-alone object file into a library is done by adding Libtool information to the configuration file used by Automake, the file Makefile.am.

Listing 8-1. *A configure.ac File for the Sample fibonacci Application*

```
AC_PREREQ(2.59)
AC_INIT(Fibonacci, 1.0, wvh@vonhagen.org)
AM_INIT_AUTOMAKE(fibonacci, 1.0, wvh@vonhagen.org)
AC_CONFIG_SRCDIR([fibonacci.c])
AC_PROG_CC
AC_HEADER_STDC
AC_CHECK_HEADERS([stdlib.h])
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Listing 8-2. *A configure.ac File with Libtool Integration*

```
AC_PREREQ(2.59)
AC_INIT(Fibonacci, 1.0, wvh@vonhagen.org)
AM_INIT_AUTOMAKE(fibonacci, 1.0, wvh@vonhagen.org)
AC_CONFIG_SRCDIR([fibonacci.c])
AC_PROG_CC
AC_PROG_LIBTOOL
AC_HEADER_STDC
AC_CHECK_HEADERS([stdlib.h])
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Listing 8-3 shows the simple Automake configuration file that was initially used in the sample application. Because no libraries are being used at this point, this file simply defines the name of the binary that will be generated by the resulting Makefile and the source code modules required in order to produce this binary, and identifies a script that can be executed in order to verify that the fibonacci application was successfully compiled.

Listing 8-3. *A Makefile.am File for the Sample fibonacci Application*

```
bin_PROGRAMS = fibonacci
fibonacci_SOURCES = fibonacci.c calc_fib.c
TESTS = run_tests
```

Listing 8-4 shows the Makefile.am file from Listing 8-3 after the addition of the macros and arguments necessary to create a library named `libfib` from the source module `calc_fib.c` instead of compiling the latter as a simple object file. This is being done solely to illustrate how to use Libtool to create a simple library as part of the build process. Creating a library from existing functions is a common occurrence when developing complex applications and is essentially a mandatory step when you want to be able to call these functions from other applications. Using libraries in complex applications also simplifies deployment and maintenance because the libraries can be updated without requiring recompilation of the main executable.

Listing 8-4. *A Makefile.am File with Libtool Integration*

```
bin_PROGRAMS = fibonacci
fibonacci_SOURCES = fibonacci.c
fibonacci_DEPENDENCIES = libfib.la
lib_LTLIBRARIES = libfib.la
libfib_la_SOURCES = calc_fib.c
fibonacci_LDADD = "-dlopen" libfib.la
libfib_la_LDFLAGS = -version-info 1:0:0
```

Comparing Listings 8-3 and 8-4 shows that a relatively small number of changes are required to convert one or more existing code modules from static objects that are directly linked to the resulting executable into a library. For this example, the following is a summary of the changes required:

- Remove the name of the source file from the existing `program_name_SOURCES` macro entry. The sources for the new library are identified in the subsequent `library_name_SOURCES` entry.
- Add the `program_name_DEPENDENCIES` macro entry to indicate that the sample fibonacci program also depends on the existence of the new library, `libfib.la`. The name used here is the name of the Libtool summary file, not the eventual name of the library that will be used on the platform where you configure and compile your application. As discussed earlier, the Libtool summary file provides platform-specific information about the library that will actually be used on the system where the application is being built. This type of indirection enables you to create platform-independent Makefile.am files.
- Define the library required by the sample application using the `_lib_LTLIBRARIES` macro entry.
- Define the name of any source files required to create the Libtool library using the `library_name_SOURCES` macro entry.
- Use the `program_name_LDADD` macro entry to define any additional arguments that need to be identified to the linker/loader when compiling the sample application. Enclosing the `-dlopen` flag within double quotes and including it here is something of a hack, but most versions of Automake do not correctly support the `program_name_LDFLAGS` variable, where you would think such specifications actually belonged. An upward-compatible solution is to specify them as additional loader arguments and to enclose the option within double quotation marks in order to protect the additional arguments from being processed. Future versions of Automake will enable you to specify linker flags in the more appropriate `program_name_LDFLAGS` variable.
- Add an example of using the `library_name_LDFLAGS` option to specify arguments passed to the linker/loader when building the library. In this case, the option and its associated value specifies the version number of the library.

Once you have modified the configure.ac and Makefile.am files for your application, you can execute the Autoconf application and run the resulting configure script to generate a Makefile for your application. For your convenience, the sidebar titled “Overview of Using Libtool with Autoconf and Automake” provides a summary of the entire Autoconf/Automake process for applications that use libraries and want to quickly and easily generate them using Libtool as part of the autoconfiguration process.

OVERVIEW OF USING LIBTOOL WITH AUTOCONF AND AUTOMAKE

The general steps required to create Autoconf and Automake configuration files that automatically invoke Libtool to build the libraries necessary for an application are as follows:

1. Create the configure.ac file for your application as described in the Chapter 7 section "Creating configure.ac Files," adding the `AM_INIT_AUTOMAKE` macro, as discussed in the Chapter 7 section "Creating Makefile.am Files and Other Files Required by Automake." Add the `AC_PROG_LIBTOOL` macro after the `AC_PROG_CC` macro. For simple applications, the final configure.ac file should look something like the one shown in Listing 8-2.
2. Run the `aclocal` script to create a local file of `m4` macro definitions (such as the definition of the `AM_INIT_AUTOMAKE` macro) for use by Autoconf.
3. Create the Makefile.am file for your application as described in the Chapter 7 section "Creating Makefile.am Files and Other Files Required by Automake." For simple applications that use a single library, the final Makefile.am file should look something like the one shown in Listing 8-4.
4. Execute the Automake program with the `--add-missing` command-line option to copy in any required files that are currently missing from your distribution but for which templates are provided as part of your Automake installation. This step also identifies any files that you must create manually. If you do not need or want to create these auxiliary information files, you can use the touch program to create empty placeholders for these files.
5. Execute the `libtoolize` script with the `--add-missing` command-line option to copy in any files required by Libtool that are currently missing from your distribution but for which templates are provided as part of your Libtool installation.
6. Run the Autoconf program to generate the Makefile.in file used by Automake and automatically run Automake to generate the Makefile for your application.
7. Run the resulting configure script to generate the Makefile for your application and then use the `make` or `make install` command to actually build or build and install your application.

Troubleshooting Libtool Problems

Libtool has been used in thousands of development projects and is really quite robust at this point. The section "Command-Line Modes for Libtool Operation" showed some of the more common errors that are easy to make when using Libtool from the command line. This section discusses a few others.

One very common error is trying to link object files and Libtool shared-library objects, as in the following example:

```
$ libtool --mode=link gcc -g -O -o libfib.la calc_fib.o
```

```
*** Warning: Linking the shared library libfib.la against the non-libtool
*** objects  calc_fib.o is not portable!
```

While not fatal on systems that support shared libraries, this is indeed a fatal error on systems that do not support shared libraries. A more appropriate (and correct) Libtool command would be the following:

```
$ libtool --mode=link gcc -g -O -o libfib.o calc_fib.o
```

```
/usr/bin/ld -r -o libfib.o calc_fib.o
```

Another common error results from trying to link existing external libraries in an unknown format with Libtool libraries that may be in another format. This is commonly the case if you are linking your application with libraries that are provided in object or archive format by a software vendor. The theory is that you can simply link these libraries into your application, but this may conflict with your system's default values for using shared or static libraries. In most cases, the easiest way around this problem is to explicitly create your application using static libraries, since this is the "lowest common denominator" of library file formats.

The Libtool documentation, discussed in the next section, describes some other common problems and provides suggestions for working around or eliminating them.

Getting More Information About Libtool

As discussed earlier in this chapter, Libtool is a GNU project that is hosted at `http://www.gnu.org`. For this reason, the central source of information about Libtool is the Libtool home page at `http://www.gnu.org/software/libtool/libtool.html`.

If you are very interested in the GNU autoconfiguration tools Autoconf and Automake, an excellent book is available that also contains a fair amount of information about Libtool: *GNU Autoconf, Automake, and Libtool*, Gary Vaughan, Ben Elliston, Tom Tromey, and Ian Lance Taylor (New Riders, 2000. ISBN: 1-57870-190-2). The authors and publisher were kind enough to make this book available under the Open Publication License, so you can browse and download the book online, if you wish. The URL for the book's home page is `http://sources.redhat.com/autobook`. You can download the book in a variety of formats at `http://sources.redhat.com/autobook/download.html`. Although online documentation is convenient, I strongly encourage you to buy a copy of this book if you are doing serious work with any of the GNU autoconfiguration tools; the authors deserve your support.

CHAPTER 9

Troubleshooting GCC

As the world's most popular set of compilers, GCC has been compiled and configured for almost every modern computer system that has the resources to execute the compilers that it provides. As a set of cross-compilers, it has also been configured for many systems that can't actually run those compilers.

GCC's C and C++ compilers are the standard compilers for these languages nowadays. The new Fortran and Java compilers are exciting, and the Ada and Objective C compilers provide stable support for these languages on almost any platform. GCC's ability to function as a cross-compiler makes it easy to compile C and C++ applications that run on one platform but generate applications targeted for another, as explained elsewhere in this book. This capability even opens up GCC to being used to generate code for systems that themselves may not have the memory or disk-space resources required to run it natively. I have used GCC to build applications for everything from minicomputers to PDAs, and have thus experienced a variety of interesting occurrences that some would call bugs or broken installations. I prefer to think of them as "learning experiences."

In this chapter, I'll pass on the fruits of these experiences. This chapter is designed to help you deal with known problems that exist in GCC when used on various platforms, platform- or implementation-specific details of GCC that may be confusing, and common usage or installation problems. Luckily, usage and installation problems, by far the most common cause of GCC problems, are the easiest to correct.

You may never need to read this chapter, which is fine with me and, I'm sure, with the GNU GCC team at the Free Software Foundation and elsewhere. However, preventative medicine is good medicine; you may find it useful to skim through this chapter before you attempt to build or install GCC. Knowing what not to do is often as important as knowing what to do in the first place—though knowing what not to do may be a larger topic. If you experience problems installing or using GCC that fall outside the information provided in this chapter, please let me know by sending me an e-mail at the address given in the introduction to this book. I'll be happy to help you however I can and will add your experience and its solution to the pool of common knowledge. Be a good GCC Samaritan—perhaps your experience can save other GCC travelers from their own train wreck.

Note This chapter is not intended to replace the release notes for the current version of GCC, which will probably always be more up-to-date than this book can be—after all, GCC is constantly evolving, while this book reflects a moment in (recent) time. This chapter highlights known issues at the time that this book was written, so that you can identify (and avoid) potentially obscure problems these issues may cause.

Coping with Known Bugs and Misfeatures

The term *misfeatures* is one of the computer industry's favorite ways of referring to what more cynical people would describe as bugs, broken code, or broken scripts. In computer terms, a *feature* is an attractive capability that is provided by a software package; a misfeature is therefore a humorous way of identifying an unattractive capability (i.e., a problem).

The release notes and any online version-specific documentation associated with any GCC release are always the best documentation for a list of known problems with a specific release of GCC. Change information about GCC releases is found at URLs such as `http://gcc.gnu.org/gcc-VERSION/changes.html`, where *VERSION* is a major version of GCC, such as 3.4, 4.0, and so on. At the time this book was written, information about all the releases in the 4.0 family was provided on a single page, `http://gcc.gnu.org/gcc-4.0/changes.html`. Information about truly ancient releases of GCC may no longer be available online. If you cannot find information about the version of GCC that you are using, perhaps it is time for an upgrade.

Aside from release notes and change information, the various GCC-related newsgroups (discussed in Chapter 10) generally contain up-to-date problem reports, suggestions, questions, and general entertainment. Within a short while after a new GCC release, the Web becomes an excellent source of information, because most of the GCC newsgroups show up on archive sites that are then indexed by Web spiders and made available through popular search engines.

A few specific issues identified in the release notes for GCC at the time this book was written are the following:

- The fixincludes script that generates GCC local versions of system header files interacts badly with automounters. If the directory containing your system header files is automounted, it may be unmounted while the fixincludes script is running. The easiest way to work around this problem is to make a local copy of `/usr/include` under another name, temporarily modify your automounter maps (configuration files) not to automount `/usr/include`, and then restart autofs (on a Linux system), or whatever other automounter daemon and control scripts you are using. If your entire `/usr` directory structure is automounted, such as on many Solaris systems, simply put together a scratch machine with a local `/usr` directory, build GCC there, and then install it. You can then use the tar application to archive the directory and install it on any system, regardless of its automount configuration.
- The fixproto script may sometimes add prototypes for the `sigsetjmp()` and `siglongjmp()` functions that reference the `jmp_buf` datatype before that type is defined. You can resolve this problem by manually editing the file containing the prototypes, placing the `typedef` in front of the prototypes.

■ **Note** The fixincludes and fixproto scripts discussed in this section are only relevant if you are building GCC and they are provided as part of the GCC source code. If you are installing a newer version of GCC from an archive, or are simply using the version of GCC that came with your system, the problems noted in these scripts are irrelevant to you.

- If you specify the `-pedantic-errors` option, GCC may incorrectly give an error message when a function name is specified in an expression involving the C comma operator. This spurious error message can safely be ignored.

Using -### to See What's Going On

Before proceeding to the discussion of possible problems and suggested solutions, it is worth calling out my favorite GCC option for debugging any GCC problem (not including syntax and logical errors in my code, which, er, I have seen a few times). This is GCC's -### option, which causes any GCC compiler to display an extremely verbose listing of what it would do when compiling a specific application. This option is very similar to the standard –v option supported by all GCC compilers, with the exception that it does not produce any binaries—it just tells you what it would have done. Using the -### option is analogous to running the make program with the –n option, and it just verbosely reports what it would have done.

Consider the following examples. First, I will compile a standard hello, world application:

```
$ gcc -o hello hello.c
$
```

No output there, which is a good thing. That's a pretty simple program to screw up. Now, let's look at the output of the same compilation command when executed by adding the -### option to the command line:

```
$ gcc -### -o hello hello.c
```

```
Using built-in specs.
Target: x86_64-unknown-linux-gnu
Configured with: ../gcc/configure --prefix=/usr/local/gcc4.2svn --enable-threads \
    --enable-languages=c,c++,objc,fortran,obj-c++
Thread model: posix
gcc version 4.2.0 20060102 (experimental)
 "/usr/local/gcc4.2svn/libexec/gcc/x86_64-unknown-linux-gnu/4.2.0/cc1" ➥
     "-quiet" "hello.c" "-quiet" "-dumpbase" "hello.c" "-mtune=k8" "-auxbase" ➥
     "hello" "-o" "/tmp/ccoBIRAk.s"
 "as" "-Qy" "-o" "/tmp/ccpqTsUl.o" "/tmp/ccoBIRAk.s"
 "/usr/local/gcc4.2svn/libexec/gcc/x86_64-unknown-linux-gnu/4.2.0/collect2" ➥
     "--eh-frame-hdr" "-m" "elf_x86_64" "-dynamic-linker" ➥
     "/lib64/ld-linux-x86-64.so.2" ➥
     "-o" "hello" "/usr/lib/../lib64/crt1.o" "/usr/lib/../lib64/crti.o" ➥
     "/usr/local/gcc4.2svn/lib/gcc/x86_64-unknown-linux-gnu/4.2.0/crtbegin.o" ➥
     "-L/usr/local/gcc4.2svn/lib/gcc/x86_64-unknown-linux-gnu/4.2.0" ➥
     "-L/usr/local/gcc4.2svn/lib64" ➥
     "-L/lib/../lib64" "-L/usr/lib/../lib64" ➥
     "-L/usr/local/gcc4.2svn/lib/gcc/x86_64-unknown-linux-gnu/4.2.0/../../.." ➥
     "/tmp/ccpqTsUl.o" "-lgcc" "--as-needed" "-lgcc_s" ➥
     "--no-as-needed" "-lc" ➥
     "-lgcc" "--as-needed" "-lgcc_s" "--no-as-needed" ➥
     "/usr/local/gcc4.2svn/lib/gcc/x86_64-unknown-linux-gnu/4.2.0/crtend.o" ➥
     "/usr/lib/../lib64/crtn.o"
$
```

I have broken many of these lines so that they fit on the printed page, but I think that you get the idea. This sort of verbose output (without execution) makes it easy to spot configuration, installation, include, or library problems that can prevent any GCC compiler from working correctly. To correct some of these sorts of problems, you can simply adjust library or include file paths. For internal problems, you can dump the specs file and fix the errors in there; but in many cases, you may need to rebuild or reinstall your GCC compilers.

Resolving Common Problems

One of the biggest advantages of open source software is that everyone has access to the source code and can therefore put it under a microscope whenever necessary to identify and resolve problems. For GCC, quite probably the most popular open source package of all time, a parallel advantage is that you are rarely the first person to encounter a specific problem. The Web and the GCC documentation are excellent sources for descriptions of common problems encountered when using any GCC compiler and, more importantly, how to solve them.

This section lists some common problems you may run into when using GCC on a variety of platforms and in a variety of ways, and provides solutions or workarounds. It is important to recognize that the fact that one can identify common problems does not mean that GCC itself is laden with problems. As you can see from the number of command-line options discussed in appendixes A and B, and some of the advanced usage models discussed in the chapters on specific GCC compilers, the GCC compilers are some of the world's most flexible programs. The number of knobs that you can turn on GCC compilers sometime leads to problems by accidentally invoking conflicting options, environment variables settings, and so on.

Problems Executing GCC

It is very rare that a version of GCC is installed in a way that all users of a system cannot execute it. Installing GCC incorrectly could actually cause this problem, in which case you would see a message like the following when trying to execute the GCC C compiler:

```
bash: gcc: Permission denied
```

If you get this message, you will have to contact your system administrator to see if gcc was intentionally installed with restrictive file permissions or ACLs (access control lists).

More commonly, problems executing a GCC compiler on a system are related to making sure that you are executing the right version of the compiler, as explained in the following two sections.

Using Multiple Versions of GCC on a Single System

If you are running an open source operating system such as Linux, FreeBSD, NetBSD, or one of the others, there is a good chance that your system came with a version of GCC that is included as part of your operating system's default installation. This is incredibly useful for most day-to-day purposes, but you may eventually want to upgrade the version of GCC available on your system, often because a new version adds features or capabilities not present in an earlier version.

Having multiple versions of GCC installed on a single system is not usually a problem. As explained in Chapter 11 and Appendix A, each version of GCC is consistent and contains internal references to the location where it was intended to be installed. The GCC C compiler binary (gcc) is primarily a driver program, using internal specifications to determine the options with which to run the programs associated with each stage of the compilation process (the C preprocessor, linker, loader, and so on). However, the GCC C compiler binary itself (gcc) currently contains hard-coded strings that identify the directory where that version of gcc expects to find these programs.

Unfortunately, gcc's internal consistency cannot guarantee that you are executing the "right" version of gcc. If you have multiple versions of the GCC compilers installed on your system and they were all installed using the default names, make sure that your `PATH` environment variable is set so

that the directory where the version you are trying to execute is listed before any directory containing another version of GCC. You can also execute any GCC compiler by using its full pathname, as in `/usr/local/gcc41/bin/gcc`. If installed correctly, each GCC binary will execute the versions of all the other programs used during the compilation process that were built at the same time as that version of gcc.

Setting the value of your `PATH` environment variable is done differently depending on the command shell that you are running. If you are using the Bourne-Again shell (bash), you can set the value of this variable by using a command such as the following:

```
$ export PATH=new-directory:${PATH}
```

Replace `new-directory` with the full pathname of the directory containing the version of GCC that you actually mean to run, as in the following example, which puts the directory `/usr/local/bin` at the beginning of your existing search `PATH`:

```
$ export PATH=/usr/local/bin:${PATH}
```

If you are using the C shell (csh) or the TOPS-20/TENEX C shell (tcsh), the syntax for updating the `PATH` environment variable is slightly different.

```
% setenv PATH /usr/local/bin:${PATH}
```

■ **Tip** On most Linux and other Unix-like systems, you can find all versions of GCC that are currently present in any directory in your `PATH` by executing the `whereis gcc` command, which will return something like the following:

```
$ whereis gcc
gcc: /usr/bin/gcc /usr/local/tivo-mips/bin/gcc /usr/local/bin/gcc
```

You can then use the `which gcc` command to identify which one is located first in your `PATH`, and adjust your `PATH` as necessary.

Problems Loading Libraries When Executing Programs

Encountering messages such as the following when trying to execute a program that you just built is a common source of frustration:

```
/foo: error while loading shared libraries: libfoo-0.9.so.21: ➥
    cannot open shared object file: No such file or directory
```

This isn't really GCC's fault—instead, what this means is that the application uses shared libraries and the loader can't locate a shared library that the application requires. The most common solution for this is to investigate where the specified library is installed, and to make sure that this directory is listed in the text file /etc/ld.so.conf. This file contains a list of the directories that the loader should search for shared libraries—one directory per line. If you have to add a new directory to this file, you must also run the `ldconfig` command after updating /etc/ld.so.conf to ensure that the loader's in-memory cache of where to look for shared libraries has been updated.

Most applications that build shared libraries display a message stating where they have been installed, as in the following example:

```
----------------------------------------------------------------------
Libraries have been installed in:
   /usr/local/somelib

If you ever happen to want to link against installed libraries
in a given directory, LIBDIR, you must either use libtool, and
specify the full pathname of the library, or use the `-LLIBDIR'
flag during linking and do at least one of the following:
   - add LIBDIR to the `LD_LIBRARY_PATH' environment variable
     during execution
   - add LIBDIR to the `LD_RUN_PATH' environment variable
     during linking
   - use the `-Wl,--rpath -Wl,LIBDIR' linker flag
   - have your system administrator add LIBDIR to `/etc/ld.so.conf'

See any operating system documentation about shared libraries for
more information, such as the ld(1) and ld.so(8) manual pages.
----------------------------------------------------------------------
```

In this case, you would need to add the directory `/usr/local/somelib` to the file /etc/ld.so.conf and then run `ldconfig`. Both of these operations require root privileges. If you do not have root privileges on the system where you encounter this problem, you can follow the advice of the second suggestion in the message and add `/usr/local/somelib` to your `LD_LIBRARY_PATH` environment variable, as in the following example:

```
$ export LD_LIBRARY_PATH=/usr/local/somelib:$LD_LIBRARY_PATH
```

When building applications for your system, you should always check the build logs or output window for messages of the sort shown previously, and make sure that you've added any new library locations to /etc/ld.so.conf on systems where you have root privileges (this is the last suggestion in the bullet list in the message shown earlier in this section). You can test whether adding a specific directory to your library path will resolve shared library problems by manually executing a command such as the following, where `/full/path/to/new/shared/library/dir` is the full path to the directory containing the new shared library, and *ProgramName* is the name of your application:

```
$ /lib/ld.so.version --library-path /full/path/to/new/shared/library/dir ProgramName
```

You can then run this command multiple times, using the `--verify` and `--list` options to ensure that the right directories are being searched and that the right libraries are being found.

You can, of course, eliminate shared library problems in general by linking your application statically. You can do this by adding the `--static` flag to the `CFLAGS` used during compilation (if you are building C applications), but this increases the size of your binary and eliminates both the application size and the library update advantages of using shared libraries.

'No Such File or Directory' Errors

Seeing a "No such file or directory" error when trying to execute a binary that you know is present (such as /bin/ls) can be maddening. This is an interesting twist on the "foo: command not found" error that you get when you try to execute a command that does not actually exist.

Luckily, the cause and solution are usually quite simple—this error indicates a problem with the shared library loader, usually /lib/ld-linux.so.X or /lib/ld.so.X. The most common cause of this problem is a bad symbolic link to the actual loader, which is usually the file /lib/ld-linux.X.Y.Z.so. This error often occurs due to a failed upgrade to a new version of Glibc or incorrectly configured

symlinks in a root filesystem that you created. If the BusyBox binary (busybox) is present on the system where this error occurs, you can usually use BusyBox to resolve the problem as described in the section of Chapter 12 titled "Using BusyBox to Resolve Upgrade Problems." If this error occurs in a root filesystem that you constructed on another system, you can correct the problem using that system's native utilities and then rebuild the root filesystem with the correct links.

Problems Executing Files Compiled with GCC Compilers

After successfully compiling a program using a GCC compiler, the message "cannot execute binary file" is depressing in its simplicity. Luckily, its cause is usually quite simple. This message typically means that the binaries produced by the version of GCC that you executed are actually targeted for a different type of system than the one that you are running on. This usually means that you are running a version of GCC that was configured as a cross-compiler, which runs on one type of system but produces binaries designed to run on another.

Most cross-compilers are built and installed using prefixes that identify the type of system on which they are designed to run. For example, the version of gcc named ppc8xx-linux-gcc is designed to run on an x86 system but produces binaries that are intended to execute on systems with PowerPC 8xx–family processors. However, for convenience' sake, people often create aliases or symbolic links named *gcc* that point to a specific cross-compiler. If you see the message "cannot execute binary file," try the following:

- Use the `alias gcc` command to check that you are not picking up a bad alias or the alias for another compiler. If you see a message like "alias: gcc: not found," this is not the problem.
- Verify that the binary produced by gcc (for example) is actually compiled for the architecture on which you are running by executing the `file filename` command. If its output does not match the type of system on which you are running, you may have accidentally invoked a version of gcc that was built as a cross-compiler for another type of system. You can also execute the compiler with the `--dumpmachine` option (e.g., `gcc –dumpmachine`) to identify the type of system that your version of gcc is producing output for.
- Verify which version of gcc (or other compiler) you are actually running by executing a command such as `which gcc`. In this case, this would return the first version of gcc that is currently found in your path. Make sure that this is the version of gcc that you actually want to run. If it is not, you can adjust the value of your `PATH` environment variable so that the directory containing the version that you want to run is found first, as described in the previous section "Using Multiple Versions of GCC on a Single System."

Running Out of Memory When Using GCC

This problem is uncommon and is usually only seen when using GCC compilers on System V–based systems with broken versions of `malloc()`, such as SCO Unix. In general, I hope that you are not using SCO Unix for anything; but if you are stuck with it for some reason, the easiest solution is to provide a working (i.e., nonsystem) version of `malloc`, the Linux/Unix memory allocation function. The GNU C Library, Glibc, includes a version of `malloc()` that you can build stand-alone. Other popular `malloc()` replacements are the Hoard memory allocator (`http://www.cs.umass.edu/~emery/hoard`), which is especially efficient in multiprocessor environments, and Wolfram Gloger's `ptmalloc()` implementation (`http://www.malloc.de/en`). After building and installing whichever `malloc()` you have chosen, you can relink GCC, specifying the name of your new version of `malloc()` using a command-line argument such as the following:

```
MALLOC=/usr/local/lib/libgmalloc.a
```

■ **Tip** If you have installed and built GCC on your system, you can simply recompile the file gmalloc.c and pass it as an argument when relinking your GCC compilers:

```
MALLOC=gmalloc.o
```

Moving GCC After Installation

To be blunt, **do not attempt to move GCC and associated programs to another directory after building and installing them.** It is always better to rebuild GCC than to try to trick it into running from a directory other than the one for which it was built.

■ **Note** This section generally applies to using GCC on actual Linux systems. If you are using Cygwin, GCC is built so that it can be moved to other directories (i.e., it is relocatable).

As mentioned previously, the GCC C compiler binary (gcc) is primarily a driver program, using spec files to determine how to run the programs associated with each stage of the compilation process (the C preprocessor, linker, loader, and so on). However, gcc itself currently contains hard-coded strings that identify the directory where that version of GCC expects to find these programs. Patches are available for GCC to make it possible to move a prebuilt version of GCC by dumping the spec files, editing them, and then configuring the compiler to use the updated spec files; but this is an accident waiting to happen. Do you really need to do this?

If the gcc binary cannot find one or more of the programs or libraries that it requires, you will see messages such as the following whenever you try to compile a program using GCC.

```
gcc: installation problem, cannot exec 'cc1': No such file or directory
```

This specific message means that the gcc binary was unable to find the program cc1, which is the first stage of the C preprocessor, and is located in gcc's library directory, gcc-lib. If you are truly out of space somewhere, you can move gcc's library directory to another location and then create a symbolic link from the correct location to the new one. For newer versions of GCC, this directory consumes around 35MB of disk space, which is not a huge amount of space by today's standards.

The worst thing about moving something and using symbolic links to glue things back together is that you will almost certainly forget that you have done this at some point. This makes it easy to delete a random directory somewhere, forgetting that symbolic links elsewhere point to it. Unless your backup strategy is a lot more robust than mine, you will almost certainly mess yourself up by doing this. The unfortunate thing about breaking your current GCC installation is that it prevents you from building another version of GCC, forcing you to resort to downloading and installing precompiled binaries in package or gzipped tar file formats.

General Issues in Mixing GNU and Other Toolchains

Most GCC users find that the GCC toolchain is a complete solution for all of their compilation needs. However, there may be times when you need to mix the GCC compilers with components of the compilation food chain that are provided by third parties. This is often the case when using optimized tools from third parties, or when you yourself are developing an enhanced preprocessor or other

part of the compiler toolchain. The following are general suggestions for resolving problems when trying to integrate third-party tools with GCC:

- Check the release notes and documents for both the foreign toolchain and the version of GCC that you are using, in order to identify any known interoperability issues. The next section of this chapter discusses some specific known problems.
- Search the Web for information about known interoperability problems.
- Verify that the problem is actually where you think it is. If you are using a version of the make program other than GNU make, ensure that your version of make supports integrating other tools and passing mandatory command-line options to them correctly. If you are not using GNU make and your version of the make program does not do this correctly, perhaps you should consider performing a free upgrade for your existing make program by downloading and installing GNU make. You can execute the `make –dry-run` command to see a detailed list of what is being passed to the GCC compiler that you are using. You should also make sure that you're executing the version of make that you intended to execute. For example, some Cygwin systems have multiple versions of make installed.
- Verify that the GCC compiler that you are using can actually find the application or library that you are trying to integrate. You can display any GCC compiler's idea of the current search path by using the `--print-search-dirs` option, as in `gcc --print-search-dirs`. GCC may either not be finding a required library or may be finding the wrong one.
- Verify that the GCC compiler that you are using is executing the right subprograms and finding the correct libraries. For example, you could determine what version of the library library.a your compiler is finding by adding `--print-file-name=library.a` to the `CFLAGS` that you're using during compilation, if compiling C applications. As a similar example, you could determine what version of the cc1 subprogram your gcc compiler is executing by adding `--print-prog-name=cc1` to the `CFLAGS` that you are using during compilation.
- If you are replacing portions of the GCC toolchain with commercial or other tools and experience problems, first make sure that you are correctly integrating the two sets of tools. All commercial toolchains support their own sets of command-line options and may invoke subcomponents (such as an assembler or preprocessor) with options that you may not be aware of because they are supplied internally.
- Use verbose modes for both GCC and whatever tools you are integrating with it. You may be able to spot missing command-line options that you can then pass to the appropriate component of the toolchain by using the appropriate version of GCC's `-X` option. GCC compilers provide the `-Xa` option to pass specific arguments to the assembler, the `-Xl` option to pass arguments to the linker, and the `-Xp` option to pass options to the preprocessor. Once you have found a combination that works for you, you can encapsulate these options by adding them to the contents of your Makefile or shell's `CFLAGS` environment variable.
- If you cannot use primary GCC command-line options to "do the right thing" when invoking commercial or other non-GCC toolchain components, you may be able to modify environment variables in your Makefile to add new command-line options to those passed to the appropriate tool during the compilation process. If you are using GNU make, each of the components in the GCC toolchain (preprocessor, assembler, linker) has a corresponding `FLAGS` environment variable (`ASFLAGS`, `LDFLAGS`, `CPPFLAGS`) that can be set in your Makefiles in order to specify command-line options that are always supplied to the appropriate tool. If you are not using GNU make, your version of the make program does not support these flags. If you are using GCC, perhaps you should consider upgrading your existing make program to GNU make.

■ **Tip** If you modify your Makefile, you can execute the `make -n` command to show the commands that would be executed to recompile your application without actually executing them.

- If the version of the make program that you are using does not support modifying the flags that you are passing to each phase of the compilation process, it may still enable you to set Makefile variables for the tools themselves. If so, you can often include specific sets of options in those application definitions. For example, GNU make supports Makefile environment variables that identify the assembler (`AS`) and the C preprocessor (`CXX`). You may be able to set these variables to the name of the binary for your third-party software plus whatever command-line options are necessary to cause your third-party software to "play nice" with GCC.
- If you are still experiencing problems, talk to the vendor. They may not be happy to know that you want to use their expensive software in conjunction with a free tool, but you have paid for support. Depending on where the problem seems to appear, you can use some combination of the GCC compilers' many debugging options (`-X`, `-c`, `-save-temps`, and `-E` come to mind) to preserve temporary files or halt the compilation process at the point at which the problem materializes. For example, if your third-party assembler cannot correctly assemble code generated by gcc, you can halt gcc's compilation process just before entering the assembler, or simply preserve the temporary input files used by the GNU assembler and provide that to your vendor so they can attempt to identify the problem.
- Make sure that the third-party tool that you are using conforms to the same C or C++ standards as the GCC defaults. See Appendix A for a discussion of using the `-std=`*`STD`* command-line option to specify a different standard for GCC to conform to during compilation.
- The GCC's C++ compiler uses a different application binary interface (ABI) than other C++ compilers. Similarly, GCC's C++ compiler does not do name mangling in the same way as other C++ compilers. For these reasons, object files compiled with another C++ compiler cannot be used with object files compiled using G++. You must recompile your entire application using one or the other. Though this may cause you a bit of work, it was done intentionally to protect you from more subtle problems related to the internal details of your C++ compiler's implementation. If G++ used standard name encoding, programs would link against libraries built using or provided with other compilers, but would randomly crash when executed. Using a unique name-encoding mechanism enables incompatible libraries to be detected at link time, rather than at runtime.
- Similarly, make sure you are not using GNU C or C++ extensions that other tools may not recognize.

The preceding is a general list of interoperability problems. The next section provides specific examples of interoperability and integration problems that are discussed to varying degrees in the GCC documentation itself.

Specific Compatibility Problems in Mixing GCC with Other Tools

This section lists various difficulties encountered in using GCC together with other compilers or with the assemblers, linkers, libraries, and debuggers on certain systems. The items discussed in this section were all listed in the documentation provided with various releases of GCC. If you are experiencing problems using GCC on one platform but not another, make sure that you check GCC's Info file for up-to-date information. You can run Info by using the command `info gcc` after installing GCC. Using Info is explained in detail in Appendix C.

The following are some known interoperability problems on specific platforms:

- On AIX systems, when using the IBM assembler, you may occasionally receive errors from the AIX assembler complaining about displacements that are too large. You can usually eliminate these by making your function smaller or by using the GNU Assembler.
- On AIX systems, when using the IBM assembler, you cannot use the dollar sign in identifiers even if you specify the `-fdollars-in-identifiers` option due to limitations in the assembler. You can resolve this by not using this symbol in identifiers, or by using the GNU Assembler.
- On AIX systems, when compiling C++ applications, shared libraries and dynamic linking do not merge global symbols between libraries and applications when you are linking with `libstdc++.a`. A C++ application linked with AIX system libraries must include the `-Wl,-brtl` option on the linker command line.
- On AIX systems, when compiling C++ applications, an application can interpose its own definition of functions for functions invoked by `libstdc++.a` with "runtime linking" enabled. To enable this to work correctly, applications that depend on this feature must be linked with the runtime-linking option discussed in the previous bullet item and must also export the function. The correct set of link options to use in this case is `-Wl,-brtl,-bE:exportfile`.
- On AIX systems, when compiling NLS-enabled (national language support) applications, you may need to set the `LANG` environment variable to `C` or `En_US` in order to get applications compiled with GCC to link with system libraries or assemble correctly.
- AIX systems do not provide weak symbol support. C++ applications on AIX systems must explicitly instantiate templates. Symbols for static members of templates are not generated.
- Older GDB versions sometimes fail to read the output of GCC versions 2 and later. If you have trouble, make sure that you are using an up-to-date version of GDB (version 6.1 or better). You'll be happier about a newer version of GDB in general.
- If you experience problems using older debuggers such as dbx with programs compiled using GCC, switch to GDB. The pain of the GDB learning curve is much less than the pain of random incompatibilities—plus it is a good career investment.
- If you experience problems using profiling on BSD systems, including some versions of Ultrix, the static variable destructors used in G++ may not execute correctly or at all.
- If you are using a system that requires position-independent code (PIC), the GNU Assembler does not support it. To generate PIC code, you must use some other assembler (usually /bin/as on a Unix-like system).
- If you are linking newly compiled code with object files produced by older versions of GCC, you may encounter problems due to different binary formats. You may need to recompile your old object files in order to link successfully.
- The C compilers on some SGI systems automatically expand the `-lgl_s` option into `-lgl_s -lX11_s -lc_s`. This may cause some IRIX programs that happened to build correctly to fail when you first try to recompile them under GCC. GCC does not do this expansion—you must specify all three options explicitly.
- On SPARC systems, GCC aligns all values of type `double` on an 8-byte boundary and expects every `double` to be aligned in the same fashion. The Sun compiler usually aligns `double` values on 8-byte boundaries, with the exception of function arguments of type `double`. The easiest way around this problem without extensive code changes is to recompile your entire application with GCC. The GCC documentation provides sample code to work around this problem, but patching a torn pair of pants does not make them new—plus you have to maintain the patches.

- On Solaris systems, the `malloc` function in the libmalloc.a library may allocate memory that is only aligned along 4-byte boundaries. Unfortunately, GCC on SPARCs assumes that doubles are aligned along 8-byte boundaries, which may cause a fatal signal if doubles are stored in memory allocated by Sun's libmalloc.a library. The only easy solution to this problem is to use `malloc` and related functions from libc.a rather than using the libmalloc.a library.

Problems When Using Optimization

The key problem with optimization is that sometimes it works too well. When using optimization, there will always be a certain amount of disagreement between an optimized executable and your source code. For example, you may find that local variables cannot be traced or examined when you are debugging an optimized application. This is often caused by the fact that GCC may have optimized the variable out of existence.

In general, you rarely need to use optimization when you are still in the debugging phase of application development. If an application works correctly before optimization but no longer works correctly when you specify optimization levels such as `-O2` or `-O3`, you should make sure that you are actually `malloc`ing every data structure that you access through a pointer. Nonoptimized programs may accidentally work if they access memory that is actually associated with other data structures that are not currently being used. By minimizing allocation and eliminating unnecessary or unused structures, optimization may cause "working" programs to fail in this circumstance.

Problems with Include Files or Libraries

As part of the process of building GCC, GCC runs shell scripts that make local copies of system header files that GCC believes exhibit various types of problems. Most target systems have some number of header files that will not work with GCC because they have bugs, are incompatible with ISO C, or are designed to depend on special features provided by other compilers. The best known of the scripts run by GCC is called fixincludes, for fairly obvious reasons.

After making local copies of these files and updating them, GCC uses these updated system header files instead of the generic system header files. If you subsequently install updated versions of your system header files, an existing GCC installation does not know that the original files have changed. GCC will therefore continue to use its own copies of system header files that may no longer be appropriate to the kernel that your system is running or the libraries that the operating system is currently using.

To resolve this sort of problem, you can either regenerate GCC's header files or (to use the really big hammer) reinstall GCC and cause it to regenerate its local copies of the system header files.

- To simply regenerate GCC's header files, run the mkheaders script, which is installed in the directory `install-root/libexec/gcc/target/version/install-tools`.
- To reinstall GCC and cause it to regenerate its header files, change to the directory in which you built GCC and delete the files named stmp-fixinc and stmp-headers, and the entire `include` subdirectory. You can then rerun the `make install` command, and GCC will recreate its local copies of problematic header files. If this process generates errors, you will need to obtain a newer copy of GCC. Your operating system vendor may have made changes that are extensive enough to make them incompatible with the shell scripts provided with your current version of GCC.

■ **Note** On some older operating systems, such as SunOS 4, header file directories contain machine-specific symbolic links in certain places. This makes it possible to share most of the header files among hosts running the same version of SunOS 4 on different machine models. However, this also means that the directory of fixed header files is good only for the machine model on which it was built. For the most part, this is not an issue because only kernel-level programs should care about the differences between different types of machines. If absolutely necessary, you can build separate sets of fixed header files for various types of machines, but you will have to do this manually. Check the Web for information about scripts that various old-time Sun users have created to do this.

Working with existing system libraries is very different from working with text files such as include files. GCC attempts to be a conforming, freestanding compiler suite. However, beyond the library facilities required by GCC internally, the operating system vendor typically supplies the rest of the C library. If a vendor's C library does not conform to the C standards, programs compiled with GCC compilers that use system libraries may display strange warnings, especially if you use an option such as `-Wall` to make your GCC compiler more sensitive. If you see multiple warnings, and existing applications compiled with the vendor's C compiler exhibit strange behavior when compiled with gcc, you should strongly consider using the GNU C library (commonly called Glibc), which is available as a separate package from the GNU Web site (`http://www.gnu.org/software/libc/libc.html`). Glibc provides ISO C, POSIX, BSD, System V, and X/Open compatibility for Linux systems (and for the GNU project's own HURD-based operating system). Chapter 12 of this book can help you get Glibc installed and working on your system. If you do not want to switch to this or are not running Linux or HURD, your only other alternative is to petition your operating system vendor for newer libraries. You have my sympathy in advance.

Mysterious Warning and Error Messages

The GNU compiler produces two kinds of diagnostics: errors and warnings. Each of these has a different purpose:

- Errors report problems that make it impossible to compile your program. GCC compilers report errors with the source filename and the line number where the problem is apparent.
- Warnings report other unusual conditions in your code that may indicate a problem, although compilation can (and will) proceed. Warning messages also report the source filename and the line number, but include the leading text string "warning" to distinguish them from error messages.

Warnings indicate points in your application that you should verify. This may be due to using questionable syntax, obsolete features, or nonstandard features of GNU C or C++. Many warnings are only issued if you specify the "right" one of GCC's `-W` command-line options (or the `-Wall` option, which turns on a popular selection of warnings).

GCC compilers always try to compile your program if this is at all possible. However, in some cases, the C and C++ standards specify that certain extensions are forbidden. Conforming compilers such as gcc or g++ must issue a diagnostic when these extensions are encountered. For example, the gcc compiler's `-pedantic` option causes gcc to issue warnings in such cases. Using the stricter `-pedantic-errors` option converts such diagnostic warnings into errors that will cause compilation to fail at such points. Only those non-ISO constructs that are required to be flagged by a conforming compiler will generate warnings or errors.

To increase the gcc's tolerance for outdated or older features in existing applications, you can always use the `-traditional` option, which attempts to make gcc behave like a traditional C compiler, following the C language syntax described in the book *The C Programming Language, Second Edition*, Brian Kernighan and Dennis Ritchie (Prentice Hall, 1988. ISBN: 0-131-10362-8). The next section discusses differences and subtleties between the gcc and K&R C compilers (Kernighan and Ritchie represent the *K* and *R* in K&R).

Incompatibilities Between GNU C and K&R C

This section discusses some of the more significant incompatibilities between GNU C and K&R C. In real life, the chances of encountering these problems are small, unless you are recompiling a huge legacy application that was written to work with a K&R C compiler, or, if like the author of this book, you still automatically think of ANSI C as "that new thing."

■ **Note** As mentioned in the previous section, using gcc's `-traditional` command-line option causes gcc to masquerade as a traditional Kernighan and Ritchie C compiler. This emulation is necessarily incomplete, but it is as close as possible to the original.

Some of the more significant incompatibilities between GNU C (the default standard used by gcc) and K&R (non-ISO) versions of C are the following:

- The gcc compiler normally makes string constants read-only. If several identical string constants are used at various points in an application, gcc only stores one copy of the string. One consequence of this is that, by default, you cannot call the `mktemp()` function with an argument that is a string constant, because `mktemp()` always modifies the string that its argument points to. Another consequence is that functions such as `fscanf()`, `scanf()`, and `sscanf()` cannot be used with string constants as their format strings, because these functions also attempt to write to the format string. The best solution to these situations is to change the program to use arrays of character variables that are then initialized from string constants. To be kind, gcc provides the `-fwritable-strings` option, which causes gcc to handle string constants in the traditional, writable fashion. This option is automatically activated if you supply the `-traditional` option.
- To gcc, the value `-2147483648` is positive because `2147483648` cannot fit in an `int` data type. This value is therefore stored in an unsigned `long int`, as per the ISO C rules.
- The gcc compiler does not substitute macro arguments when they appear inside of string constants. For example, the following macro in gcc:

  ```
  #define foo(a) "a"
  ```

  will produce the actual string `"a"` regardless of the value of `a`. Using the `-traditional` option causes gcc to do macro argument substitution in the traditional (old-fashioned) non-ISO way.
- When you use the `setjmp()` and `longjmp()` functions, the only automatic variables guaranteed to remain valid are those declared as volatile. This is a consequence of automatic register allocation. If you use the `-W` option with the `-O` option, gcc will display a warning when it thinks you may be depending on nonvolatile data in conjunction with the use of these functions. Specifying gcc's `-traditional` option causes gcc to put variables on the stack in functions that call `setjmp()`, rather than in registers. This results in the behavior found in traditional C compilers.

- Programs that use preprocessing directives in the middle of macro arguments do not work with gcc. ISO C does not permit such a construct and is poor form anyway. The `-traditional` option will not help you here.
- Declarations of external variables and functions within a block apply only to the block containing the declaration—they have the same scope as any other declaration in the same place. Using the `-traditional` option causes gcc to treat all external declarations as globals, as in traditional C compilers.
- In traditional C, you can combine type modifiers with preexisting `typedef`ed names, as in the following example:

  ```
  typedef int foo;
  typedef long foo bar;
  ```

  In ISO C, this is not allowed. The `-traditional` option cannot change this type of gcc behavior because this grammar rule is expressed in high-level language (Bison, in this case) rather than in C code.
- The gcc compiler treats all characters of an identifier as significant, rather than only the first eight characters used by K&R C. This is true even when the `-traditional` option is used.
- The gcc compiler does not allow whitespace in the middle of compound assignment operators such as +=.
- The gcc compiler complains about unterminated character constants inside of a preprocessing conditional that fails if that conditional contains an English comment that uses an apostrophe. The `-traditional` option suppresses these error messages, though simply enclosing the comment inside comment delimiters will cause the error to disappear. The following is an example:

  ```
  #if 0
      You cannot expect this to work.
  #endif
  ```

- Many older C programs contain declarations such as `long time()`, which was fine when system header files did not declare this function. However, systems with ISO C headers declare `time()` to return `time_t`. If this is not the same size as `long`, you will receive an error message. To solve this problem, include an appropriate system header file (<time.h>) and either remove the local definitions of `time()` or change them to use `time_t` as the return type of the `time()` function.
- When compiling functions that return structures or unions, gcc's output code normally uses a method different from that used on most versions of Unix. As a result, code compiled with gcc cannot call a structure-returning function compiled with an older compiler such as PCC (Portable C Compiler), and vice versa. You can tell gcc to use a compatible convention for all functions that return structures or unions by specifying the `-fpcc-struct-return` option.

Abuse of the __STDC__ Definition

Though more a stylistic or conceptual problem than a gcc issue, the meaning of `__STDC__` is abused frequently enough that it deserves a short discussion here.

Programmers normally use conditionals on whether `__STDC__` is defined in order to determine whether it is safe to use features of ISO C such as function prototypes or ISO token concatenation. Because vanilla gcc supports all the features of ISO C, all such conditionals should test as true, even when the `-ansi` option is not specified. Therefore, gcc currently defines `__STDC__` as long as you do not specify the `-traditional` option. The gcc compiler defines `__STRICT_ANSI__` if you specify the `-ansi` option or an `-std` option specifying strict conformance to some version of ISO C.

Unfortunately, many people seem to assume that `__STDC__` can also be used to check for the availability of certain library facilities. This is actually incorrect in an ISO C program, because the ISO C standard says that a conforming freestanding language implementation should define `__STDC__`, even though it does not have the library facilities. The command `gcc -ansi -pedantic` is a conforming freestanding implementation, and it is therefore required to define `__STDC__`, even though it does not come with an ISO C library.

■ **Tip** You should not automatically undefine `__STDC__` in a C++ application. Many header files are written to provide prototypes in ISO C but not in traditional C. Most of these header files can be used in C++ applications without any changes if `__STDC__` is defined. If `__STDC__` is undefined in a C++ application, they will all fail and will need to be changed to contain explicit tests for C++.

Resolving Build and Installation Problems

Having been successfully installed and used on a huge number of platforms for the past 15 years or so, GCC's build and installation processes are quite robust. GCC requires a significant amount of disk space during the build and test process, especially if you build and test all of the compilers (C, C++, Ada, Java, Fortran) that are currently included in the GNU Compiler Collection. One of the most common problems encountered when building and installing the GCC compilers is simply running out of disk space.

Before you begin to build and install GCC compilers, think about the tools that you are using to build them. For example, it is easier and safer to build GCC using GNU make rather than any system-specific version of make. Similarly, GCC is best built with GCC. Though this seems like a paradox, it really is not. When bootstrapping GCC on Solaris boxes without the official (paid) Solaris C compiler, I usually download a prebuilt binary of the gcc C compiler for Solaris (from a site such as `http://www.sunfreeware.com`), download the latest GCC source code, and then use the former to build the latter. Maybe I'm just paranoid.

■ **Note** Using a compiler to compile itself is an impressive feat, though conceptually an equally impressive exercise in recursion.

If you have previously built and installed GCC and then encounter problems when rebuilding it later, make sure that you did not configure GCC in your primary directory unless you intend to always build it there. The GCC build and installation documents suggest that you create a separate build directory and then execute the configure script from that directory as `../gcc-VERSION/configure`. This will fail if you have previously configured GCC in the primary directory.

As mentioned earlier, a common error encountered when building GCC on NFS-dependent systems is that the fixincludes script that generates GCC local versions of system header files may not work correctly if the directory containing your system header files is automounted—it may be unmounted while the fixincludes script is running. The easiest way to work around this problem is to make a local copy of `/usr/include` under another name, temporarily modify your automounter maps (configuration files) not to automount `/usr/include`, and then restart autofs (on a Linux system) or whatever other automounter daemon and control scripts you are using. If your entire `/usr` directory structure is automounted, such as on many Solaris systems, you should just put together a scratch machine with a local `/usr` directory and build your GCC compilers there. After installing it, archive the installation directory and install your GCC compilers onto other systems from the archive.

Building and installing GCC has been done by thousands of users everywhere. I have built and used GCC on everything from Apollo workstations (back in the day) and a huge variety of embedded systems, all the way up to the 64-bit home Linux boxes that I use today. If you encounter a problem, check the release notes, the README file, and the installation documentation. If you still have problems, feel free to send me e-mail at the address given in the introduction to this book, or visit `http://gcc.gnu.org`, or check the news archives at `http://www.google.com`. If you have experienced a problem, chances are that someone else has too.

CHAPTER 10

Additional GCC and Related Topic Resources

As the most widely used compiler on computer systems today, GCC has a tremendous number of users. It is therefore not surprising that there is a similarly large number of online resources where you can obtain detailed information about GCC, ask questions, read others' questions, share your solutions and expertise, and so on. Many Web sites, mailing lists, and Usenet newsgroups are dedicated to sharing information about GCC, its development status, known problems, bug fixes, and the like.

One caveat that you should remember about the information that you obtain through any free resource (such as the Web, Usenet, etc.) is that you should take it with a grain of salt. Luckily, if you are having a problem using GCC and someone proposes a workaround or solution, it is usually easy enough to test the suggestion and determine whether it resolves the problem that you are having.

This chapter provides an overview of the most popular online sources of information about GCC, explaining how to find them, how to access them, and so on. The Web is a living, breathing place—between the time that this chapter was written and the time you are reading it, many new, information-packed sites may have appeared. Regardless, the sites in this chapter are a good starting point; many of them (such as the GCC-related Usenet newsgroups) have been available for years and will continue to be excellent sources of information in the future.

Usenet Resources for GCC

Long before the commercial availability of the Internet, Unix users developed a worldwide distributed discussion system known as Usenet. Usenet, also often referred to as *netnews*, consists of a flexible set of newsgroups with names that are classified hierarchically by subject. Messages are read from and posted to these newsgroups by people using software generally known as newsreaders, though Web browsers such as SeaMonkey, Netscape, and Internet Explorer contain built-in software for reading and sending messages to Usenet newsgroups. Sending a message to a newsgroup is generally referred to as posting a message to that newsgroup.

In order to access Usenet news, you must specify a news server in your newsreader software or Web browser. A news server is a computer system you have access to that serves as a repository for Usenet newsgroups. Most news servers restrict access to their copies of various newsgroups based on your network address, host name, or some similar mechanism.

After specifying a news server in your browser or newsreader software, you can display a list of the newsgroups that the server keeps a copy of, and you can then subscribe to any of the newsgroups that you are interested in. When you subsequently use your browser or newsreader software to view the newsgroup, a list of all messages posted to each newsgroup since the last time you checked is displayed.

Posting messages to a Usenet newsgroup works much like e-mail. There are two basic types of newsgroups: moderated and unmoderated. Posts made to moderated newsgroups are screened by individuals known as *moderators*, who determine if a message is appropriate to a given newsgroup and submit it to the newsgroup. Unmoderated newsgroups are open—they can be posted to by anyone. It is the responsibility of the person posting a message to only post messages on relevant subjects to these newsgroups. Unmoderated newsgroups therefore have a much higher percentage of spam than moderated newsgroups (which have none if correctly moderated). But messages appear on them much more quickly than on moderated newsgroups because the latter require verification of each message.

When you post a message to an unmoderated newsgroup, or once a message has been approved for posting on a moderated newsgroup, the news server you are connected to adds a copy of that message to its local repository of messages. It also forwards your message to other news servers it is configured to feed messages to. Long ago, traffic between Unix news servers was done over telephone lines using a protocol called Unix to Unix Copy Protocol (UUCP), which is specified in Internet RFC 976, but the majority of this traffic is now done over the Internet using Network News Transport Protocol (NNTP), which is specified in Internet RFC 977.

Usenet newsgroups not only provide a valuable (though perhaps overwhelming) source of information on a variety of topics, they can also be just plain fun. Over 99,000 Usenet newsgroups are available on topics ranging from GCC to highly polarized political and sexual topics. The number of Usenet newsgroups that you have access to depends on the number carried by the news server that you select.

■ **Note** For the most part, the Usenet newsgroups associated with GCC receive the same messages that are exchanged on the GCC mailing lists discussed in the next section. One significant advantage of using Usenet newsgroups is that they are always available from a large number of sites, are not subject to transient e-mail outages, and serve as a long-term repository for information that is almost always available somewhere on the Web.

Most Internet service providers (ISPs) make a news server available to their customers. If yours does not, for whatever reason, a number of open news servers are available on the Internet. These freely accessible news servers enable anyone to access the newsgroups that they host, but are often restricted to just letting people read Usenet news or post messages to Usenet newsgroups. You can obtain an up-to-date list of these by doing a Web search for "netnews public server." At the time of this writing, one of the best open news servers for reading Usenet news was newscache0.freenet.de, which provides read-only access to more than 23,000 newsgroups. Similarly, one of the best open news servers for posting messages to Usenet newsgroups is available at news.wplus.net. This open news server enables you to post messages to more than 62,000 newsgroups.

Selecting Software for Reading Usenet News

Web browsers such as SeaMonkey, Netscape, and Internet Explorer all include software for reading Usenet news, as does e-mail software such as Pine. Google provides a great Web-based front end for reading news in your browser at `http://groups.google.com`. Also, a large number of applications are designed specifically for reading Usenet news, many of which are available as open source. Some of the most common open source newsreaders are the following:

- *Agent*: This popular commercial newsreader from Forte Inc. runs on Windows systems. A scaled-down but free version of Agent, known as Free Agent, is also available from Forte. For more information, or to download or purchase a copy, see Forte's Web site at `http://www.forteinc.com/agent/`.
- *Knews*: This is an open source graphical newsreader that uses the X Window System. Knews has an easy-to-use, intuitive interface and supports displaying message threads (related sets of messages on a common topic). You can obtain source code or binary versions of Knews from its home page at `http://www.matematik.su.se/~kjj/`.
- *NewsXpress*: This is a popular newsreader that is available free and is designed for Windows 95 and Windows NT systems. It is available at `http://www.malch.com/nxfaq.html`.
- *slrn*: The "s-lang read news" newsreader is a terminal-oriented open source newsreader that runs on Mac OS X, Windows, and Unix/Linux systems. You can obtain source code or binary versions of slrn from its home page at `http://slrn.sourceforge.net/`.
- *TIN*: The Tass + Iaian's Newsreader is an open source terminal-oriented newsreader that is quite popular and has an incredible number of options available for fine-tuning its behavior and performance. You can obtain the source code for the latest versions at its home page, `http://www.tin.org/`.

Summary of GCC Newsgroups

The following is a list of the primary GCC-related newsgroups that are available via Usenet and their descriptions. For more information about GCC-related mailing lists, see the next section, "Mailing Lists for GCC."

- *gnu.gcc*: Due to the hierarchical nature of Usenet newsgroups, this newsgroup is primarily intended as a container for the other GCC newsgroups. You should not post to this list—though you will occasionally see posts appear there, they should actually go into one of the other newsgroups discussed in this section.
- *gnu.gcc.announce*: This newsgroup is intended for distributing announcements and progress reports on GCC. This is a moderated list that is not intended for general discussion, but is restricted to posts made by GCC maintainers. As such, this is a low-traffic list. This newsgroup contains the same messages that are distributed through the info-gcc mailing list.
- *gnu.gcc.bug*: This newsgroup is intended for posting and viewing bug reports for GCC, fixes for reported problems, and suggestions for future improvements to GCC. This is a moderated list. If you are working on GCC, you can post fixes in the form of patches applied to some version of GCC. When posting patches, you should also include sample code that illustrates the problem that your patch resolves. GCC is actively under development all over the world, and it may be that your fix is subsumed in someone else's—in which case your test code will help demonstrate the correctness of any other patches. When preparing a test case, the most generally accepted form is output from the C preprocessor (cpp) that can be passed directly to the first phase of the GCC C compiler (cc1). It is also best to provide standard C code (rather than C++ or Objective C), since this reduces the chance that the problem that you are fixing actually occurs in a preprocessing phase of the compiler. This newsgroup contains the same messages that are distributed through the bug-gcc mailing list. This is a low-traffic newsgroup.

- *gnu.gcc.help*: This newsgroup is intended as a forum where people using and installing GCC can ask general GCC questions or ask for help with specific issues. It is not a forum for reporting problems—those should be posted to the gnu.gcc.bug newsgroup or the bug-gcc mailing list. The difference between a help request and a problem report can be subtle: basically, if GCC does not build on your system, if it does not execute correctly once installed, or if a command-line option does not do what it is supposed to, those sorts of problems should be reported as bugs. (Make sure that you have read this book and the documentation before reporting something as a bug!) This newsgroup contains the same messages that are distributed through the help-gcc mailing list. This is a high-traffic newsgroup.
- *pilot.programmer.gcc*: This newsgroup is intended for use by people who are developing applications for Palm PDAs using GCC. The original Palms were known as Palm Pilots, hence the name of the newsgroup. Developing applications on platforms with limited memory is a challenge in the first place, and the Palm OS provides some interesting challenges in terms of segmenting applications so that they load and execute correctly. This is a low-traffic newsgroup.

Though there are other groups whose names contain the string "gcc" (such as linux.act.gcc, linux.dev.gcc, list.linux-activists.gcc, and pocunix.mail.linux.gcc, among others), these are generally ghost lists that were created at one time but no longer receive any significant traffic. However, aside from the GCC-specific lists mentioned in the previous section, there are other Usenet newsgroups that contain relevant information that is useful to GCC users. Some of the most useful related lists are the following:

- *gnu.g++.announce*: Analogous to gnu.gcc.announce, this is a moderated newsgroup that is intended for distributing announcements and progress reports on g++, the C++ compiler provided as part of the GNU Compiler Collection. This is a low-traffic list where any news is good news. This newsgroup contains the same messages that are distributed through the info-g++ mailing list.
- *gnu.g++.bug*: Analogous to gnu.gcc.bug, this is a moderated newsgroup where you can report problems with g++ and the g++ debugger gdb+, submit fixes, and propose suggestions for future versions of the g++ compiler. This newsgroup contains the same messages that are distributed through the bug-g++ mailing list. This is a low-traffic list.
- *gnu.g++.help*: Analogous to gnu.gcc.help, this is an unmoderated list where you can ask general questions about g++ or request help with specific problems. This newsgroup contains the same messages that are distributed through the help-g++ mailing list. This is a high-traffic list.
- *gnu.g++.lib.bug*: Analogous to gnu.glibc.bug, this is a moderated list where you can report problems with the g++ library (/usr/lib/libstdc++.so.<version>), submit fixes, and propose suggestions for future extensions to the library. This newsgroup contains the same messages that are distributed through the bug-lib-g++ mailing list. This is a low-traffic list.
- *gnu.gdb.bug*: Analogous to gnu.gcc.bug, gnu.g++.bug, and gnu.g++.lib.bug, this is a moderated list where you can report problems with the GNU debugger, gdb; submit fixes; and propose suggestions for future enhancements to gdb. This newsgroup contains the same messages that are distributed through the bug-gdb mailing list. This is a low-traffic list.
- *gnu.glibc.bug*: Analogous to gnu.g++.lib.bug, this is a moderated list where you can report problems with the GNU C library (libc-<version>.so), submit fixes, and propose suggestions for future extensions to the library. This newsgroup contains the same messages that are distributed through the bug-glibc mailing list. This is a low-traffic list.

Usenet newsgroups provide a convenient, classic way of viewing posts, questions, and responses made by others, and for posting your own questions to unmoderated lists. Accessing Usenet newsgroups to retrieve this information is especially convenient in locations where ISPs charge for e-mail. There are

a variety of repositories on the Internet (see http://www.livinginternet.com/u/uu_arch.htm for a list) where old newsgroups are archived, providing long-term access to information that would otherwise be somewhat transient. Due to the number of posts made to Usenet newsgroups, most news servers periodically delete posts that are older than a specified period that is defined in each news server's configuration files.

If you do not have access to a Usenet news server, have no limitations on the amount of e-mail that you can receive, and want the immediate participation provided by e-mail, you may want to join one or more of the GCC-related mailing lists. The various GCC-related mailing lists are discussed in the next section.

Mailing Lists for GCC

Mailing lists are analogous to the Usenet newsgroups discussed in the previous section, except that posts made to a list are sent to a central address and are then directly forwarded to all of the subscribers to the list. Posts made to most mailing lists can be received one-by-one, as fast as they can be forwarded to subscribers by the list server, or in digest form, where all of the posts made each day are collected and sent as a single daily e-mail message.

Aside from the existence of most of the utilities that all Linux and most *BSD distributions depend on, one indication of the tremendous amount of software that the Free Software Foundation has created and enhanced over the years is the huge number of mailing lists that the Free Software Foundation hosts. Almost every package and GNU utility has its own mailing list, though the amount of traffic on each list varies with the popularity of the utility or package, the amount of changes and number of open defects associated with the software, and whether the list is moderated. Open, unmoderated mailing lists, such as those where users can ask for help or information on using a specific utility, tend to get a substantial amount of traffic.

You can subscribe to the basic GNU mailing lists through the Web site at http://mail.gnu.org/mailman/listinfo/. However, due to their sheer number, the GCC lists are hosted through a different mechanism. To subscribe to the GCC lists, visit the URL http://gcc.gnu.org/lists.html and scroll down the page until you find the subscription form. This form provides a drop-down list of available GCC-related lists and enables you to specify whether you want to receive posts one-by-one or in digest format. After clicking the Process That! button, the list will send confirmation e-mail to the e-mail subscription address that you specified. You can generally simply reply to this message and your subscription to the list will be accepted.

Posting to any of the GCC-related lists is as simple as sending e-mail to *listname*@gcc.gnu.org, where *listname* is the name of the list that you want to post your message to. The names of the lists hosted by http://gcc.gnu.org/ that you can post to are explained in the next section, "GCC Mailing Lists at gcc.gnu.org." For information about how to be a good party member and not irritate any of the people on the lists (or frustrate yourself), see the section later in this chapter titled "Netiquette for the GCC Mailing Lists."

After subscribing to any GCC mailing list, you can unsubscribe by sending mail to the unsubscribe address listed in the confirmation e-mail that you receive. Information about unsubscribing is also provided on the Web page at http://gcc.gnu.org/lists.html.

GCC Mailing Lists at gcc.gnu.org

This section lists the various GCC-related mailing lists that are hosted by http://gcc.gnu.org for discussion and related activities regarding GCC, its components, and associated software packages. Each entry explains the type of information that a list is intended to provide, and highlights any relationship between these mailing lists and the Usenet newsgroups discussed in the previous section.

The `http://gcc.gnu.org/` site hosts two general classes of mailing lists: those intended for public consumption and those that are primarily targeted for internal use by GCC developers, people porting GCC to other platforms, and GCC maintainers. Although all of the mailing lists hosted at `http://gcc.gnu.org/` can be read by anyone, only the lists intended for public consumption can also be posted to by members of the general public. Because the internal lists are targeted toward a relatively small group of focused developers, members of the general public cannot post to these lists. This makes them easier for their intended audience to use, and also eliminates the chance that they are bombarded by spam.

The next two sections summarize the available mailing lists in each of these categories.

Read/Write Mailing Lists

This section lists the GCC mailing lists that are hosted at `http://gcc.gnu.org/` and can be both read and posted to by members of the general public. At the time of this writing, these open mailing lists included the following:

- *fortran*: An open list for discussing the use and development of the Fortran language front end of GCC and its corresponding runtime library, libgfortran. Patches to gfortran and libgfortran should go to both this list and to the gcc-patches list.
- *gcc*: An open, high-volume list for discussing GCC development and testing issues that are not specifically relevant to any of the other GCC lists discussed in this section. This list is also used for announcing and discussing major changes to the GNU Compiler Collection itself, such as abandoning ports to specific systems or architectures, abandoning specific front ends to GCC, and so on. This newsgroup is essentially a GCC newspaper, keeping you abreast of important topics and trends.
- *gcc-announce*: A moderated, low-volume list where GCC maintainers post announcements about releases or other important events.
- *gcc-bugs*: An open, high-volume list where users can file problem reports, submit fixes, and generally discuss unresolved issues in GCC. The posts made to this mailing list are also forwarded to the gnu.gcc.bugs mailing list, but are moderated there.
- *gcc-help*: An open, high-volume list that provides a forum where people can ask for assistance in building and using GCC. All of the posts made to this mailing list are also forwarded to the gnu.gcc.help Usenet newsgroup.
- *gcc-patches*: An open, high-volume list to which GCC developers can post and discuss patches to GCC. These patches can apply to any aspect of GCC, from patches to front ends, patches to the GCC core, and even patches to and corrections for the GCC Web pages.
- *gcc-testresults*: An open, moderate-volume list where users of GCC can post test results for different versions of GCC on any platform.
- *java*: An open, high-traffic list for discussing the Java language front end for GCC (gcj), the runtime library associated with this front end (libgcj), and the development of both. Patches to these should not be posted to this list—instead, they should be posted to the java-patches list.
- *java-announce*: A moderated, low-volume list where the maintainers and developers of the Java language front end or runtime library for GCC post announcements about releases or other important events.

- *java-patches*: An open, medium-traffic list for submitting and discussing patches to the Java language front end to GCC and its associated runtime library. Patches to gcj and libgcj should be submitted to both this mailing list and to the standard gcc-patches mailing list.
- *libstdc++*: An open, high-traffic list for discussing the standard C++ library (libstdc++-v3) and for posting patches to this library. Patches to this library should be sent to both this list and the gcc-patches mailing list.

Read-Only Mailing Lists

As mentioned previously, some of the GCC mailing lists hosted at `http://gcc.gnu.org/` are intended for a somewhat limited audience and/or are automatically posted to by automated systems such as CVS (Concurrent Versions System), a source code maintenance and tracking system that is widely used on Unix, Linux, and *BSD systems. As such, they can only be read by the general public and cannot be posted to by most people. This section lists the GCC mailing lists that can be read only by members of the general public.

At the time of this writing, the read-only mailing lists hosted at `http://gcc.gnu.org/` were the following:

- *gccadmin*: A medium-volume list to which output from nightly cron jobs run on the system `gcc.gnu.org` is posted.
- *gcc-cvs*: A relatively high-volume list that tracks source code check-ins to the GCC CVS repository. For more information about CVS, see its home page at `http://www.nongnu.org/cvs/`.
- *gcc-cvs-wwwdocs*: A relatively low-volume list that tracks check-ins to the GCC Web pages portion of the GCC CVS repository.
- *gcc-prs*: A relatively high-volume list that tracks problem reports as they are entered into the GNATS database that is used to track these issues. GNU GNATS is an excellent open source bug tracking and reporting system. GNATS facilitates problem report management and supports communication with users in a variety of different ways. Each GNATS instance stores problem reports in its own databases and provides tools for querying, editing, and maintaining these databases. For more information about GNATS, see the GNATS home page at `http://www.gnu.org/software/gnats/`. An excellent open source Web interface to GNATS, Gnatsweb, is available at `http://ftp.gnu.org/pub/gnu/gnatsweb/`.
- *gcc-regression*: A medium-volume list where the results from running regression tests on the GCC compilers are posted.
- *java-cvs*: A relatively high-volume list that tracks check-ins to the Java language compiler and runtime portions of the GCC CVS repository. Like the java-prs mailing list, a separate mailing list is provided for these messages because some GCC developers and maintainers are primarily interested in Java, and this makes it easier for them to find relevant posts. Because the Java front end to GCC is just one of a variety of front ends, messages posted to this list are also sent to the standard gcc-cvs mailing list.
- *java-prs*: A relatively high-volume list that tracks Java-related problem reports as they are entered into the GCC GNATS database. A separate mailing list is provided for these problem reports because some GCC developers and maintainers are primarily interested in Java, and this makes it easier for them to find relevant posts. Because the Java front end to GCC is just one of a variety of front ends, messages posted to this list are also sent to the standard gcc-prs mailing list.

- *libstdc++-cvs*: A relatively low-volume list that tracks source code check-ins to the libstdc++-v3 portion of the GCC CVS repository. Because this is a component of the entire GCC CVS source tree, the messages posted to this list are a proper subset of those that are posted to the gcc-cvs mailing list.

As you would expect from a dynamic, open source project that is as widely used as GCC, other mailing lists were available in the past whose contents have subsequently been rolled into one of the mailing lists discussed in this and the previous section. Of these, the one that was perhaps best known was the libstdc++-prs mailing list dedicated to problem reports regarding the C++ library used with the GCC front end for C++ and g++. This list is no longer active. Any outstanding problem reports for libstdc++ have been rolled into the standard GCC GNATS database. Old postings to this list are still available at `http://gcc.gnu.org/` as an archive—the list itself is no longer active or supported.

For additional information about using any of the GCC mailing lists, send a blank e-mail message to *`listname`*`-help@gcc.gnu.org`.

Netiquette for the GCC Mailing Lists

The GCC mailing lists are provided as a community service by the folks at the Free Software Foundation. As such, you should follow a few simple rules to ensure that your messages get there and are well-received.

The first rule of posting to one of the GCC lists is to not post off-topic messages. When posting to one of these lists, make sure that your message is relevant to the intent of the list. Similarly, you should refrain from sending the same message to multiple lists. Broadcasting your post across multiple lists (known as *cross-posting*) will probably bring you more derision than it will answers.

A second, equally important rule is that posts to the GCC lists must adhere to the spirit of open source software. Nothing will fill your mailbox with e-mail flames faster than posting a commercial message on one of the GCC lists. For example, I would be committing cultural and e-mail suicide if I posted a recommendation for this book there. Free software is free and the source code for it is freely available and must be redistributed. This book, though it is about GCC, is a commercial item. Although it is completely relevant and valuable to any GCC user, posting a note about it on any of the GCC lists would just be wrong.

A third basic rule is to only post messages in text form. Many of the people who subscribe to these lists do not use mail clients that support HTML, XML, or any other bells-and-whistles message format. Text messages are the lowest common denominator and are therefore the preferred format—they can be displayed and read in any e-mail client. Text messages are the preferred format for pure text content, such as code examples or patches.

When posting to the GCC-related mailing lists, you should try to keep your messages under 25,000 characters so that they pass successfully through all Internet mailers. That is a lot of typing, but a relatively small amount of error output if you are experiencing a problem. A better approach is to post a message containing an extract of a problem report and then send the complete output to anyone who is willing or able to help you. The GCC lists themselves have a hard limit of 100K per post, which means that you can post messages larger than 25K, but you still have no guarantee that such posts will successfully make it through the chain of mail servers necessary to reach `http://gcc.gnu.org/`.

■ **Note** The gcc-prs list, where summaries of problem reports are posted, accepts messages up to 2MB in size, but it is not an open list.

Other GCC-Related Mailing Lists

Though the majority of the GCC-centric mailing lists are hosted at `http://gcc.gnu.org`, other GCC-related mailing lists are hosted at other sites. These generally do not discuss specific GNU GCC packages, but instead focus on specific ports or applications of GCC that are not directly sponsored by the Free Software Foundation. The most interesting of these are the following:

- *COBOL for GCC mailing lists*: Though most people think of GCC primarily in terms of its C, C++, and Java compilers, the GCC core can be used as a compiler for many other programming languages. One of the more interesting of these is COBOL, not only because COBOL is a language that many people only associate with older computer systems, but also because COBOL is still one of the most widely used languages in the computer industry. Three different COBOL for GCC mailing lists are hosted at `http://sourceforge.net/`: cobolforgcc-announce, a list for announcements about COBOL for GCC; cobolforgcc-devel, an unmoderated list for discussing the development of COBOL for GCC; and cobolforgcc-users, an unmoderated list for discussing COBOL for GCC. To post to any of these lists, send e-mail to *`listname`*`@lists.sourceforge.net`, replacing *listname* with the name of the list that you want to post to. You can subscribe to, unsubscribe from, or view the archives of any of these lists at `http://sourceforge.net/mail/?group_id=5709`.
- *GCC for Palm OS mailing lists*: Conceptually related to the pilot.programmer.gcc newsgroup discussed in the previous section, the GCC for Palm OS mailing lists are intended for use by Palm programmers who are using the GCC-based PRC tools (Palm resource tools) to cross-compile applications for Palm PDAs. Three different GCC for Palm OS lists are hosted at SourceForge.net: prc-tools-announce, an unmoderated list for release information and announcements related to the PRC tools; prc-tools-cvs, a read-only list that gives changes to the CVS source code archive for the PRC tools; and prc-tools-devel, a high-traffic list for discussing enhancements and posting patches to the PRC tools. To post to any of these lists, send e-mail to *`listname`*`@lists.sourceforge.net`, replacing *listname* with the name of the list that you want to post to. You can subscribe to, unsubscribe from, or view the archives of any of these lists at `http://sourceforge.net/mail/?group_id=4429`.

World Wide Web Resources for GCC and Related Topics

As you would expect, the best source for an entire spectrum of GCC information is the primary GCC Web site at `http://gcc.gnu.org/`. This page contains introductory information and a summary of recent GCC news and announcements. The remainder of the site is organized into sections providing general information about GCC, links to a variety of documentation, links to download locations, general information about GCC development, and links for reporting or perusing defect reports (e.g., bugs). The GCC site is nicely organized and is compatible with both graphical Web browsers and text-oriented browsers such as ELinks, Links, and Lynx. If you are using a VT100 for your Web browsing, I recommend using the Links or ELinks browsers, as they do a superior job of rendering tables. The Links browser's home page is at `http://artax.karlin.mff.cuni.cz/~mikulas/links/`. The ELinks browser's home page is at `http://elinks.or.cz/`.

In recent years, wikis have become increasingly popular. A *wiki* is a special kind of Web site that allows visitors to easily add and edit content, facilitating the collaborative development of documentation, and is therefore well-suited for online projects such as open source projects. A GCC wiki is located at `http://gcc.gnu.org/wiki`, which is a great source of up-to-date information about GCC, but is not designed to be a place where you can ask questions—the mailing lists discussed earlier in this chapter are still the right place for those.

In the face of the popularity of instant messaging software, similar software that has been used on the Internet for years known as Internet Relay Chat (IRC) is enjoying a resurgence. IRC clients support long-term communication on specific subjects by creating specific *channels* associated with those topics. Channel names are prefixed by a hash mark. For example, you can find information about an IRC channel dedicated to discussing GCC development at `http://gcc.gnu.org/wiki/GCConIRC`. As with the GCC wiki, this is not a general location where you can ask general usage questions—those should still be posted on the appropriate GCC mailing lists.

Given the popularity of GCC, there are a number of other sites that provide relevant and useful information about GCC and compiler technology in general. The Web is a transient environment for information, but the resources listed in this section have been around for a while and will (hopefully) still be available if you ever need to consult them. One of my favorite general sites is The Compilers Resources Page (`http://www.bloodshed.net/compilers/`). This page is a general resource for a variety of compilers, including GCC. It lists all known free compilers and provides a mechanism where you can add others to the list if you are aware of one that is missing. It also provides links to a number of compiler construction toolkits, articles, and tutorials on compiler technology, and links to other compiler-related sites. Though this Web site is an excellent general resource for compilers and compiler technology, the only GCC-specific information available on this page is a link to the GCC home page and links to several Windows-based ports of GCC.Given that the Web is virtually infinite in size and scope, hundreds of other sites are available that provide information about building, using, and solving problems with GCC. A few seconds with your favorite search engine should result in enough links for a few days of browsing.

Information About GCC and Cross-Compilation

As discussed in Chapter 14, one of the most popular ways of using GCC is as a cross-compiler that executes on one platform yet generates binaries that will execute on another. This is especially popular when developing applications for embedded systems, where the target hardware may not have sufficient processing power or memory to support local compilation. Some good sites for Linux cross-compiler information are the following:

- The sourceware.org Web site, `http://www.sourceware.org/lists.html`, supports the crossgcc mailing list and also maintains a FAQ and an archive of old posts to this list at `http://www.sourceware.org/ml/crossgcc/`.
- Dan Kegel's site, `http://kegel.com/crosstool/`, is a good source of information about using his crosstool program to build cross-compilers, and provides a good deal of information about building cross-compilers in general.
- IBM has a great white paper about building GCC cross-compilers, which you can register for and retrieve through `https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=dw-l-cross&S_TACT=105AGX&S_CMP=GR&ca=dgr-lnxw09CrossCompiler`. You can also see LinuxDevices.com's announcement of this white paper at `http://www.linuxdevices.com/news/NS8742143554.html`.
- The LinuxDevices.com Web site contains announcements of many new GCC cross-compilers and also provides general information about building and using cross-compilers for embedded development.

Many other Internet sites provide information on specific cross-compilers and about specific situations, such as building cross-compilers for Microsoft Windows and Cygwin that will produce binaries for different embedded Linux targets.

Information About Alternate C Libraries

As discussed in Chapter 13, you can use GCC with a number of different C libraries, most of which have been created to provide smaller runtime footprints for embedded applications. Typically, you decide which standard C library to use when building GCC, but you can coerce GCC to use another. All of these C libraries have their own home pages and associated sets of information. The home pages for each of the most commonly used C libraries on Linux platforms are the following:

- *dietlibc*: `http://www.fefe.de/dietlibc/`
- *Glibc*: `http://www.gnu.org/software/libc/libc.html`
- *Newlib*: `http://sources.redhat.com/newlib/`
- *uClibc*: `http://www.uclibc.org` (also see the Buildroot site at `http://www.buildroot.org`)

For more information on alternate C libraries and using them with GCC, see Chapter 13.

Publications About GCC and Related Topics

Of course, I recommend this book as a publication that tells you all you need to know about installing and using GCC, but a number of other publications on GCC are also available. The majority of these are published by GNU Press, the publishing arm of the Free Software Foundation. The primary Web site for GNU Press is at `http://www.gnupress.org/`. You can contact the GNU Press via e-mail at `press@gnu.org`, by phone at 617-542-5942 (ask for GNU Press), and also via snail mail at the following address:

GNU Press
c/o Free Software Foundation
51 Franklin St, Fifth Floor
Boston, MA 02110-1301
USA

The following list describes other publications that are available on GCC and related topics:

- *Debugging with GDB: The GNU Source-Level Debugger*, Richard M. Stallman et al. (GNU Press, 2002. ISBN: 1-882114-88-4): After a compiler, a debugger is a developer's best friend, and the GNU debugger, GDB, has more debugging bells and whistles than you can shake a symbol at. This is yet another book that is not specifically about GCC, though it discusses the compilation requirements necessary to compile applications that can be successfully debugged using GDB. The GNU debugger enables you to analyze application crashes, monitor application execution, and step through application execution, and provides both local and remote debugging capabilities. GDB was written to work closely with GCC, and supports debugging applications written in the C, C++, Java, Fortran, and assembly languages.
- *GCC—The Complete Reference*, Arthur Griffith (Osborne/McGraw-Hill, 2002. ISBN: 0-07-222405-3): This book is a good reference for building, installing, and using GCC with a variety of different languages. It also provides summary information about related topics such as GDB, make, Autoconf, and so on. I, of course, prefer the book you are reading now, but it is always handy to have multiple books on the same subject. This book lags the current release of GCC by a few revisions, but that is to be expected in the exciting, fast-paced world of GCC.

- *GNU C Library Reference Manual* (two volumes), Sandra Loosemore, et al. (GNU Press, 2001. ISBN: 1-882114-55-8): This not a GCC-specific book, but discusses the GNU implementation of the standard C libraries (Glibc), which are typically used with GCC. This book discusses the entire spectrum of interfaces available in Glibc, and provides complete reference information, including code examples. Like GCC itself, Glibc has been undergoing a fair amount of evolution and refinement over the past few years, and therefore the actual Glibc version discussed in this book (2.2.x) lags the current release by a few versions. Regardless, the information that the book provides is extremely useful and available nowhere else (unless you read the Glibc source code). This book is available online from GNU Press.
- *GNU C Programming Tutorial*, Mark Burgess and Ron Hale-Evans (available online at `http://www.crasseux.com/books`): While not GCC-specific, this tutorial on learning the C programming language uses GCC as its compilation environment. This text is intended for beginning programmers and may therefore be too primal for existing developers looking for deep insights into GCC and C programming.
- *GNU Make: A Program for Directing Recompilation*, Richard M. Stallman, Roland McGrath, and Paul D. Smith (GNU Press, 2004. ISBN: 1-882114-83-3): Though not specific to GCC, this is an excellent book on the GNU make program that is used to incrementally compile and maintain almost every open source application (and many commercial ones). This book is an excellent resource and is well worth having, especially when using GNU make to manage the compilation of complex software projects. This book is available online from GNU Press.
- *Using and Porting GNU CC*, Richard M. Stallman (GNU Press, 1999. ISBN: 1-882114-38-8): No one knows more about GCC than its original author and the head of the Free Software Foundation, Richard Stallman. This book is an excellent book, though not necessarily light reading. The book explains how to install, run, debug, configure, and port the GNU Compiler Collection, and discusses using the C, C++, Objective C, and Fortran front ends for GCC. Given the frequency with which new versions of GCC have been released over the past year or two, the actual GCC version discussed in this book (2.95.3) lags the current release by a few versions, but the information that the book provides is still almost entirely germane. This book is available online from GNU Press.

CHAPTER 11

Compiling GCC

Compiling GCC compilers from source is widely considered a difficult or even risky undertaking. I respectfully disagree. Admittedly, the process of compiling a compiler from source is complex, but the GNU development team takes care of most of the complexity for you. Moreover, merely building a compiler imposes no risk at all. Installing it, or rather, installing it incorrectly, can certainly destabilize your system. You can, however, install the newly built compiler as a supplemental or secondary compiler rather than as the system compiler and completely sidestep this risk. On the other hand, if you follow the instructions in this chapter, you can install the latest version of GCC as your primary compiler without impacting your system's overall stability.

One of the primary points of this chapter is to demonstrate that compiling the GCC compilers from scratch is not difficult. It requires some care and attention to detail and a lot of time, but as with many tasks, good instructions make it simple to do. You can also easily build and install the latest version of GCC in another location on your system, so you can use either your system's default version of GCC or your handcrafted version. If you need some motivation to undertake the upgrade, the introduction to this book features a section titled "Why the New Edition?" that highlights the major improvements that make it worthwhile to upgrade to GCC 4 or GCC 4.x in order to get the latest bug fixes and improvements.

Note The current, official version of GCC when this book was written was version 4.1. Many of the download paths and filenames discussed later in this chapter contain version numbers that identify a specific version of GCC. The string *VERSION* is used in path and filenames to indicate where you should substitute a specific version number in order to download specific files. The main GCC Web site (`http://gcc.gnu.org`) identifies the latest released version of GCC, which is normally the one you will want to build. The instructions in this chapter will work for building any version of GCC 4.x. When it is different, information about building GCC 3.x releases is also provided.

Why Build GCC from Source?

There are a number of reasons to build GCC from source. One of the most important is that GCC's feature set (and, as a result, the pool of possible bugs) continues to grow with each new release. If you want to take advantage of new features you have two choices: wait for a precompiled binary to become available for your CPU and operating system, or build it yourself. Another important reason to build GCC yourself is that doing so enables you to customize it for your system. For example, the default configuration of precompiled GCC binaries usually includes all supported languages. If you don't need the Ada or Java front ends, for example, you can prevent them from being built, tested, and installed by passing the appropriate options to the configure script. Similarly, you can modify installation paths, munge the names of the installed binaries, or create a cross-compiler by invoking GCC's configure script using the relevant incantations.

Naturally, if you want to participate in GCC development, you will need to build GCC from source. The flow of patches that fix bugs, add new features, and extend or enhance existing features is steady, too. If a particular bug bites you or if a specific enhancement appeals to you, you will only be able to take advantage of the patch if you know how to apply it and rebuild the compiler. While I'm on the subject, allow me to point out that the act of building (and optionally testing) GCC releases, snapshots, or the latest CVS source tree constitutes participation in GCC development. How so? Simply put, without binaries to execute, you might as well use your computer for a doorstop or a bookend. In order to create binaries, you need a compiler, preferably one that is stable and efficient. GCC needs to be tested on as many different systems as possible in order to maximize its stability and efficiency and to be as bug-free as possible. If GCC builds and tests successfully on your system, that is one more data point in GCC's favor. If your particular combination of hardware and software reveals a previously unknown problem, this is better still, because either you or the developers, or you and the developers together, can identify and fix the problem.

Finally, and although some might find this peculiar, you might find successfully building a very complicated piece of software on your system to be satisfying or even entertaining. I do. Go figure.

Starting the Build Process

The process of building GCC is best approached as a series of smaller steps that organize the process into easily digestible bite-size morsels. The steps I will follow in this chapter include:

1. Verifying and satisfying software requirements
2. Preparing the installation system
3. Downloading the source code
4. Configuring the source code
5. Building the compiler
6. Testing the compiler
7. Installing the compiler

Verifying Software Requirements

Before you can successfully build GCC, you need to make sure you have at least the minimum required versions of certain utilities installed on your system. These programs and utilities must be installed before you begin building GCC, otherwise the build won't work at all, or will fail at some point in the process. If you attempt to build GCC and discover that a required utility is missing, you should install the utility and then reconfigure GCC by running the `make distclean` command in your build directory and then follow the configuration instructions given in the sections of this chapter titled "Configuring the Source Code" and "Additional Configuration Options."

Building any GCC compiler requires that the following utilities be installed on your system:

- bash (or other POSIX-compliant shell)
- GNU binutils (2.16 or better)
- Bison (only necessary for building Subversion snapshots)
- DejaGNU (necessary for running the GCC test suite)
- Flex (only necessary for building Subversion snapshots)
- gcc (or any other C-90–compliant C compiler)
- GNAT (only necessary if building the GCC Ada compiler)

- GNU make (version 3.79.1 or later)
- GNU tar (version 1.12 or later)

Building the GCC Fortran compiler requires that you have the following two math libraries installed on your system:

- GMP (GNU Multiple Precision Arithmetic Library), the latest version of which is available from `http://www.swox.com/gmp/#DOWNLOAD`
- MPFR (multiple-precision floating-point rounding), the latest version of which is available from `http://www.mpfr.org/mpfr-current/`

If you are going to be testing your compilers once you've built them (and I suggest you do), you will also need Tcl and Expect installed on your system:

- Tcl (tool command language), available from `http://tcl.sourceforge.net`
- Expect, a tool for automating interactive applications available from `http://expect.nist.gov`

If you are building a Subversion snapshot of GCC, you might also need one or more of the following tools, also available from the GNU project:

- Autoconf (`ftp://ftp.gnu.org/gnu/autoconf/autoconf-VERSION.tar.gz`)
- Automake (`ftp://ftp.gnu.org/gnu/automake/automake-VERSION.tar.gz`)
- Gperf (`ftp://ftp.gnu.org/gnu/gperf/gperf-VERSION.tar.gz`)
- Gettext (`ftp://ftp.gnu.org/gnu/gettext/gettext-VERSION.tar.gz`)

■ **Note** The filenames for the source code for these utilities use the generic value *VERSION* in download paths and filenames, indicating that you should always download and install the latest version available.

Chapter 8 describes installing Autoconf and Automake, so refer to that chapter for detailed installation instructions for these packages. To build and install any of the other utilities, use the following steps (which assume you have a working compiler):

1. Uncompress and extract the archive(s).
2. Make the extracted directory the current directory.
3. Run the configure script.
4. Execute `make`.
5. Execute `make check` (optional).
6. Execute `make install`.

Let me emphasize that Autoconf, Automake, Bison, Flex, gperf, and gettext are only necessary if you are building GCC from the Subversion tree. Prereleases, release candidates, and official releases produced by the GCC steering committee do not require these utilities.

Once you have verified that these utilities are installed (or you have installed them) on the system where you want to build GCC, the remainder of preparing the installation system consists of making sure you have sufficient disk space for the build tree and deciding whether you want to install the new compiler as the primary system compiler or as a supplemental or alternate compiler. Downloading and configuring the source code is straightforward and uncomplicated. Building the compiler is equally simple, involving one command (`make bootstrap`) and a lot of time. Strictly speaking, testing

the compiler is an optional step, but in my opinion it should be a required step. Like building GCC, testing the compiler build requires a single command and a lot of time. If the build tests out okay, and release versions should, the final step is to install the compiler.

■ **Caution** At the time this chapter was written, the GCC development team warned that GNU libc 2.2.3 and earlier should not be built with GCC 3.0 (and, accordingly, 3.1) because it introduced new changes in exception handling, resulting in a binary-incompatible Glibc. This means that programs built against a C library that was linked with an older version of GCC (2.95.3, for example) would be incompatible with a C library built with GCC 3.0 or later. The resulting incompatibility could render your system utterly unusable. Therefore, do not rebuild Glibc 2.2.3 or earlier with GCC 3.0 or later. You can use Glibc 2.2.3 with GCC 3.0 or later without any problem.

Preparing the Installation System

The GCC build process requires considerable disk space. The GCC documentation suggests that you build your GCC compilers in a directory other than the one created when you downloaded and de-archived the source code. I follow that suggestion in this book, and I refer to the directory in which you're actually building your GCC compiler(s) as the *build tree*. At the time this book was written, the archive file containing the GCC source code was 45MB in size; the unpacked source code for the latest version of GCC required almost 300MB of disk space; and the complete GCC compiler suite using the `make bootstrap` command (explained later in this chapter in the section "Compiling the Compilers") required almost 700MB of disk space. The `make bootstrap-lean` command described later in this chapter reduces the amount of disk space required, but it will always be a significant amount. Performing the `make check` step takes up an additional 20MB. So the first step to prepare your system for building the compiler is to make sure you have enough free space on the file system you will use. As a general rule of thumb, I recommend using a file system with at least 1.5GB free disk space for both the build tree and the source directory. Having more disk space than necessary is never a problem.

■ **Note** The disk usage figures mentioned here are guidelines because the actual usage fluctuates from release to release and depends on the size of the binary images, which will vary based on the operating system, architecture, and type of file system where the data is being stored.

All of the examples in this book use my home directory to hold the source code and build tree, and install the GCC compilers into subdirectories of `/usr/local`, which is their default installation location. You must therefore also make sure you have enough space on the disk or partition on which the completed compiler suite will be installed. A complete installation of all supported languages, with national language support (NLS) and with shared libraries, requires approximately 250MB. The section titled "Installing GCC" demonstrates installing the compiler into `/usr`, but the installation process described in that section also removes the existing compiler, so the net change in disk space usage is relatively small. You can use various methods to reduce the footprint of the build tree and of the installed compiler. These are mentioned in the sections titled "Downloading the Source Code" and "Configuring the Source Code."

The next step is to decide how to build and install GCC—whether as a mortal user or as a root. I recommend building it as a mortal (nonroot) user and using root privileges only to install it. GCC does not require any special access to your system during the build process, so there is no reason to build it as the superuser. If you intend to install GCC in a system directory or in a filesystem to which you do not have write access, the `make install` step must be executed by the root user or a user with root equivalence. On the other hand, if you intend to install the new GCC in your home directory, you will not need root access at all, because all of the compiler's components will be installed relative to your home directory. The examples in this chapter install GCC into a subdirectory of `/usr/local`, except in the section "Installing GCC," which discusses installation into `/usr`. Both of these installations require root access.

Downloading the Source Code

I'm going to assume you can download the source code without my help (we're all major-league gearheads here, right?). You can download the GCC source code from the GNU FTP site, or better, from a mirror that is close to you in terms of network topology. You can view a list of mirrors at `http://www.gnu.org/server/list-mirrors.html`. The files you want are

- `ftp://ftp.gnu.org/gnu/gcc/gcc-`*`VERSION`*`/gcc-`*`VERSION`*`.bz2`
- `ftp://ftp.gnu.org/gnu/gcc/gcc-`*`VERSION`*`/gcc-testsuite-`*`VERSION`*`.bz2`
- `ftp://ftp.gnu.org/gnu/dejagnu/dejagnu-`*`VERSION`*`.tar.gz`
- `ftp://ftp.gnu.org/gnu/binutils/binutils-`*`VERSION`*`.bz2`

If you are short on disk space or do not want to install the complete compiler, you can download just the core compilation engine plus the files that provide support for specific compilers. In this case, then, instead of downloading gcc-VERSION.bz2, download the compiler core, gcc-core-VERSION.bz2, and the language-specific files for each language you want.

- *Ada*: `ftp://ftp.gnu.org/gnu/gcc/gcc-`*`VERSION`*`/gcc-ada-`*`VERSION`*`.bz2`
- *C++*: `ftp://ftp.gnu.org/gnu/gcc/gcc-VERSION/gcc-g++-`*`VERSION`*`.bz2`
- *Fortran*: `ftp://ftp.gnu.org/gnu/gcc/gcc-VERSION/gcc-fortran-VERSION.bz2`
- *Java*: `ftp://ftp.gnu.org/gnu/gcc/gcc-VERSION/gcc-java-`*`VERSION`*`.bz2`
- *Objective C*: `ftp://ftp.gnu.org/gnu/gcc/gcc-VERSION/gcc-objc-`*`VERSION`*`.bz2`

You can save a small amount of disk space by leaving out the test suite, `ftp://ftp.gnu.org/gnu/gcc/gcc-VERSION/gcc-testsuite-VERSION.tar.gz`, which will also save you from having to download and install the test harness, DejaGNU (dejagnu-VERSION.tar.gz).

Installing the Source Code

After you download the various source files, create a build directory. On my system, I downloaded the source files into `~/src`, which is the `src` subdirectory of my home directory, so I created a build directory named `gcc-obj` in this directory:

```
$ cd ~/src
$ mkdir gcc-obj
```

Next, extract the tarballs:

```
$ tar -jxf gcc-VERSION.bz2
$ tar -jxf gcc-testsuite-VERSION.bz2
$ cd gcc-VERSION
```

Unless you rename the resulting directories, the source distribution extracts into a directory named `gcc-VERSION`. I'll refer to this throughout this book as the *source directory* or the *source tree* in order to distinguish the source directory from the *build tree*, which is the directory in which you compile GCC. This distinction is important because I adhere to the recommendation of the GCC developers and do not build the compiler in the source directory. Why? Keeping the source tree unpolluted by the detritus of a build makes it much easier to apply patches. A separate build directory also makes it trivial to delete an old or failed build—simply delete the entire contents of the build tree. A configuration in which the source and build directories are the same is also not as well-tested. The build process creates directories on the fly, so building in a separate directory makes sure the build process does not clobber a preexisting directory. Finally, I take the developers at their word—if they keep their source and build trees separate, I reckon it's prudent to follow their example.

■ **Caution** Do not use a build directory that is a subdirectory of the source tree. Doing so is entirely unsupported and will probably result in mysterious failures and problems.

Although the build examples in this chapter presume that you opt to install the complete collection of compilers, you are not, of course, required or compelled to do so. You can cherry-pick only the languages you want. If you prefer to use this approach and be more selective, you must at least install and build gcc-core-VERSION.bz2. This file contains the core back-end compilation engine, the C compiler, and other code common to all of the language-specific front ends. The language-specific tarballs in the following list include the front-end compiler for the given language and, for C++, Objective C, Java, and Fortran, the language-specific runtime libraries.

If you build only select languages, uncompress and extract the core compiler archive and the specific language files in the same directory. For example, to install only the core C compiler and the C++ compiler, the sequence of commands would be

```
$ tar -jxf gcc-core-VERSION.bz2
$ tar -jxf gcc-g++-VERSION.bz2
```

You can also skip the test suite, in which case you omit the step that extracts gcc-testsuite-VERSION.bz2, and do not follow the commands discussed in the section titled "Testing the Build" later in the chapter.

After this point, you should have an arrangement that resembles the following:

```
~/src
~/src/gcc-VERSION
~/src/gcc-obj
```

The source directory is `/home/wvh/src/gcc-VERSION`, where `/home/wvh/src/gcc-obj` is the build directory, and *VERSION* represents the version of GCC you downloaded and are building.

Of course, you are not obliged to work in your home directory. I do so because it avoids a lot of potential problems related to file system permissions.

Configuring the Source Code

Configuring the source code is quite simple. Make your build directory your current directory:

```
$ cd ~/src/gcc-obj
```

Invoke the configure script in the source directory, as shown in the following example:

```
$ ../gcc-VERSION/configure --prefix=/usr/local/gccVERSION
```

After a few minutes, the configuration script completes and the previously empty build directory will be configured for a GCC build. The invocation shown in the example uses the `--prefix` option to specify `/usr/local/gcc`*VERSION* (a new directory) into which the completed compiler will be installed. You should replace *VERSION* with the appropriate numeric value for the version of GCC that you are using. This enables you to keep multiple versions of the GCC compilers on your system, though you will have to make sure that the `bin` directory of the one that you actually want to use is in the list of directories in your `PATH` environment variable.

Depending on the type of compiler you intend to build, you may need to identify more precisely the system on which the compiler will run or the system for which the compiler must generate code. To do so, use the `--build=`*system*, `--host=`*system*, and `--target=`*system* arguments to configure, specifying the proper system name in *system*. The GCC build process (more generally, any modern Autoconf-based configure script) understands three different system names: the build system, the host system, and the target system. The *build system* is the machine on which you build GCC; the *host system* is the machine on which the completed compiler will run; and the *target system* is the machine for which the compiler will generate code. Perhaps surprisingly, the build, host, and target systems (or system names) do not have to be the same, and often are not.

What Is in a (System) Name?

To understand why the build, host, and target system need not always be the same system, consider how you would build GCC on a system that lacks a functioning compiler. For example, if you want to build GCC for a Microsoft Windows system, you have to have GCC running on it, right? Well, actually, you don't, because you can use a *cross-compiler*, a compiler running on one CPU architecture that generates code for another CPU architecture, neatly sidestepping a potential catch-22 situation. So, to build GCC for a Microsoft Windows–based Intel x86 system (colloquially known as a Wintel system), you could use a cross-compiler executing on a SPARC system running Solaris that knows how to create a binary that runs on a Wintel system. To compiler fans, this is known as a Canadian Cross build (`http://www.airs.com/ian/configure/configure_6.html`), which refers to some obscure joke about the Canadian parliamentary system.

Various permutations of build, host, and target names are possible. If the build, host, and target systems are all the same, you are building a *native compiler*, one that executes on and generates executables for the same type of system as the system on which it is built. Obviously, this is the simplest case. If the build and host systems are the same, but the target system is different, you are building a *cross-compiler*. If the build and target systems are the same, but the host system is different, you are building what is referred to as a *crossback compiler*, that is, you are using a cross-compiler to build a cross-compiler that will run on the system on which the compiler is being built. If the target and host systems are the same, but the build system is different, you are building a *crossed native* (or a *cross-built native* or a *host-x-host*) compiler, which means you are using a cross-compiler to build a native compiler on a third type of system. If the build, host, and target systems are all different, you are building what is called a *canadian*, a compiler that builds on one architecture, runs on another architecture, and creates code for a third architecture.

Confused? Table 11-1 shows some examples that illustrate possible combinations of build, host, and target systems.

Table 11-1. *Compiler Types*

Build	Host	Target	Compiler Type	Result
x86	x86	x86	Native	Built on an x86 to run on an x86 to generate binaries for an x86
SH	SH	ARM	Cross	Built on a SuperH to run on a SuperH to generate binaries for an ARM
x86	MIPS	x86	Crossback	Built on an x86 to run on a MIPS to generate binaries for an x86
PPC	SPARC	SPARC	Crossed native	Built on a PPC to run on a SPARC to generate code for a SPARC
ARM	SH	MIPS	Canadian	Built on an ARM to run on a SuperH to generate code for a MIPS

Why go to all this trouble? The most common reason is to bootstrap a compiler for a system that lacks one. If you do not have a running compiler for architecture A but you do have one for architecture B, you can use B's compiler to generate binaries for A, or to generate a compiler for A. To learn how to build and use a cross-compiler see Chapter 16.

Strictly speaking, you don't have to feed configure any options at all. Like most GNU software, the default configuration is to install into `/usr/local` (which is equivalent to specifying `--prefix=/usr/local`). In addition, configure script uses sane, well-tested, and generally accepted defaults for other configuration details, so you can usually rely on the compiler being compiled properly and, in general, to "do the right thing." Nevertheless, I prefer to specify a directory beneath `/usr/local` so I can have multiple installations of GCC and access them by changing the path to the gcc binary. The next section describes the configure options that let you customize GCC's compile-time defaults.

Additional Configuration Options

Even though the configure script creates a reasonable default configuration, you can exercise greater control over the configuration by calling it with various options. Table 11-2 lists the options that configure accepts, their default values, and a brief description of each option. Readers accustomed to using configure scripts are cautioned that the output from `./configure --help` might list options that are not shown in Table 11-2, as well as lack many of the options. Perversely, the `--help` option for GCC's configure script might list options that do not work, so I recommend that you don't treat the output from `--help` as the one true set of guidelines for building GCC.

Table 11-2. *Options for GCC's Configure Script*

Option	Default Value	Description
`--prefix=`*dir*	`/usr/local`	Sets *dir* as the top-level directory into which GCC will be installed
`--exec-prefix=`*dir*	`/usr/local`	Sets *dir* as the directory into which architecture-dependent files will be installed
`--bindir=`*dir*	`/usr/local/bin`	Sets *dir* as the directory into which binaries invoked by users will be installed

Table 11-2. *Options for GCC's Configure Script (Continued)*

Option	Default Value	Description
`--libdir=`*`dir`*	`/usr/local/lib`	Sets *`dir`* as the directory into which the libraries and related data used by GCC will be installed
`--with-slibdir=`*`dir`*	`/usr/local/lib`	Sets *`dir`* as the directory into which the shared library, libgcc, will be installed
`--infodir=`*`dir`*	`/usr/local/info`	Sets *`dir`* as the directory into which GCC's Texinfo (info) files will be installed
`--mandir=`*`dir`*	`/usr/local/man`	Sets *`dir`* as the directory into which the generated GCC manual pages will be installed
`--with-gxx-include-dir=`*`dir`*	`/usr/local/include/g++-v`*`ver`*	Sets *`dir`* as the default into which the g++ header files will be installed
`--program-prefix=`*`str`*	None	Adds `str` as a prefix to the names of programs installed in *`prefix`*`/bin`
`--program-suffix=`*`str`*	None	Adds `str` as a suffix to the names of programs installed in *`prefix`*`/bin`
`--program-transform-name=`*`pattern`*	None	Uses the sed script *pattern* to modify the names of programs installed in *`prefix`*`/bin`
`--with-local-prefix=`*`dir`*	`/usr/local`	Sets *`dir`* as the top-level directory into which local include files are installed
`--enable-shared[=`*`pkg`*`[,...]]`	Host	Specifies the names of libraries that should be built as shared libraries (recognized names for `pkg` are libgcc, libstdc++, libffi, zlib, boehm-gc, libjava)
`--with-gnu-as`	N/A	Tells the compiler to assume that the assembler it finds is the GNU assembler
`--with-as=`*`path`*	*`exec-prefix`*`/lib/gcc-lib/`*`target/version`*	Specifies the path to the assembler to use
`--with-gnu-ld`	N/A	Tells the compiler to assume the linker it finds is the GNU linker
`--with-ld=`*`path`*	*`exec-prefix`*`/lib/gcc-lib/`*`target/version`*	Specifies the path to the linker to use
`--with-stabs`	Host	Instructs the compiler to use BSD STABS debugging information instead of the standard format for the host system (usually DWARF2)

Table 11-2. *Options for GCC's Configure Script (Continued)*

Option	Default Value	Description
`--disable-multilib`	Varies	Prevents building multiple target libraries for supporting different target CPU variants, calling conventions, and other CPU-specific features (normally, GCC builds a default set when appropriate for the CPU)
`--enable-objc-gc`	None	(Objective-C only) Builds the Objective-C compiler with support for a conservative garbage collector, known as the Boehm-Demers-Weiser conservative garbage collector (`http://www.hpl.hp.com/personal/Hans_Boehm/gc/`), which must be built and installed before building GCC's Objective-C compiler
`--enable-threads[=`*`lib`*`]`	Target	Specifies that the target supports threading and/or the threading model supported by the target (recognized names for `lib` include `aix`, `dce`, `mach`, `no`, `posix`, `pthreads`, `rtems`, `single`, `solaris`, `vxworks`, and `win32`)
`--with-cpu=`*`cpu`*	None	Instructs the compiler to generate code for the CPU variant specified by *`cpu`*
`--enable-altivec`	None	Tells the compiler that the target supports AltiVec enhancements (only applies to PowerPC systems)
`--enable-target-optspace`	Disabled	Optimizes target libraries for code space instead of code speed
`--disable-cpp`	Disabled	Prevents installation of a C preprocessor (cpp) visible to user space
`--with-cpp-install-dir=`*`dir`*	None	Copies the C preprocessor (cpp) to *`prefix`*`/dir/cpp` and to *`prefix`*`/bin`
`--enable-maintainer-mode`	Disabled	Regenerates GCC's master message catalog, which requires the complete source tree
`--enable-version-specific-runtime-libs`	Disabled	Causes the compiler's runtime libraries to be installed in the compiler-specific subdirectory (`libsubdir`), which is useful if you want to use multiple versions of GCC simultaneously

Table 11-2. *Options for GCC's Configure Script (Continued)*

Option	Default Value	Description
`--enable-languages=`*`lang`*`[,...]`	All	Specifies the language front ends to build. See the section "Building Specific Compilers" for a list of valid values for this option with different versions of GCC.
`--disable-libgcj`	Disabled	Prevents building gcj's runtime libraries, even if building the Java front end
`--with-dwarf2`	Disabled	Instructs the compiler to use DWARF2 debugging information instead of the host default value on systems for which DWARF2 is not the default
`--enable-win32-registry`	None	Tells GCC hosted on a Microsoft Windows system to look up installation paths in the system registry, using the default *key*, `HKEY_LOCAL_MACHINE\Software\Free Software Foundation\`*key* (*key* defaults to the GCC version number)
`--enable-win32-registry=`*key*	`HKEY_LOCAL_MACHINE\Software\Free Software Foundation\`*vernum*	Tells GCC hosted on a Microsoft Windows system to look up the installation path in the system registry, using the key specified in *key*, where *vernum* is the GCC version number
`--disable-win32-registry`	Enabled only for Windows	Disables use of the Win32 registry
`--nfp`	`m68k-sun-sunosN` and `m68k-isi-bsd` only	Identifies processors that lack hardware floating-point capability
`--enable-checking`	Disabled for releases, enabled in snapshots and CVS/SVN versions	Enables extra error checking during compilation but does not change generated code
`--enable-checking=`*list*	`misc`, `tree`, `gc`	Defines the types of extra error checking to perform during compilation (possible values for *list* are `misc`, `tree`, `gc`, `rtl`, and `gcac`)
`--enable-nls`	Enabled by default	Enables national language support, which GCC uses to emit error messages and other diagnostics in languages other than American English
`--disable-nls`	Enabled by default	Disables NLS

Table 11-2. *Options for GCC's Configure Script (Continued)*

Option	Default Value	Description
`--with-included- gettext`	Disabled	Configures GCC to use the copy of GNU gettext if `--enable-nls` is specified and the build system has its own gettext library
`--with-catgets`	Disabled	Configures GCC to use the older catgets interface for message translation if `--enable-nls` is specified and the system lacks GNU gettext (ordinarily, in such a case, the build would use the copy of gettext included in the GCC distribution)
`--with-system-zlib`	Disabled	Configures the build process to use the zlib compression library installed on the build system rather than the version provided in the GCC sources

Many of the configuration options listed in Table 11-2—or their impact on the resulting compiler—require additional explanation. As a rule, the only change I make in installation directories is to use `–prefix=`*`dir`* to specify an installation directory. This enables me to install the compiler and all of its related files below the single top-level directory specified in *`dir`*, making it simple to manage multiple versions of GCC installed on a single system. The directory-manipulation options listed from `--exec-prefix=`*`dir`* to `--with-gxx-include-dir=`*`dir`* in Table 11-2 provide finer control over where the installation process drops the various compiler components during installation. A related configuration option, `--enable-version-specific-runtime-libs`, disabled by default, installs the compiler's runtime libraries into a compiler-specific subdirectory, which is again useful if you intend to use multiple versions of GCC simultaneously.

`--program-prefix` and `--program-suffix` take directory-name mangling down to the filename level. The values of *`str`* passed to these arguments are attached to the beginning or end, respectively, of the program names when the programs are installed in *`prefix`*`/bin`. For example, `--program-prefix=v3` turns gcc into v3gcc, g++ into v3g++, cpp into v3cpp, and so on. These two options make it possible to assign unique names to the user-executed binaries created during the build process. Similarly, `--program-transform-name` enables you to utterly mangle the names of files installed in *`prefix`*`/bin` by using the sed script specified in the pattern that you specify as the argument to this option. For example, specifying `--program-transform-name="s/[:lower:]/[:upper:]/g"` will translate all lowercase letters into their corresponding uppercase forms. As a result, gcc would become GCC, g++ would become G++, and so forth. You can use all three of the options discussed in this paragraph together to create a highly customized (or totally confusing) GCC installation.

The `--with-local-prefix` option does not do what you think it does. As stated in Table 11-2, it defines the top-level directory into which local include (header) files are installed. That is, it tells GCC where to find local header files, not where to install the compiler's files (use `--prefix` and kin for this purpose). So if your system keeps locally installed headers in `/opt/local/include`, use `--with-local-prefix=/opt/local`. The default value, `/usr/local`, should suffice for all Linux and most Unix and Unix-like systems.

The option `--enable-maintainer-mode` does not enable any sort of compiler voodoo or black magic known only to GCC's developers. Rather, it causes the build process to regenerate the master message catalog used by the NLS subsystem to display messages and diagnostics in non-English (well, non-American English) languages.

NLS-Related Configuration Options

Finally, here is how to make sense of the NLS-related configuration options. If you do not need non-English messages and compiler diagnostics, specify `--disable-nls`. If you need NLS and have the GNU gettext library installed, specify `--enable-nls`. If you need NLS and do not have GNU gettext installed, specify `--enable-nls` and `--with-included-gettext`. If you need NLS, do not have gettext installed, and do not want to use the copy of gettext shipped with the GCC sources, specify `--enable-nls` and `--with-catgets`.

Building Specific Compilers

You can specify different compilers using the `–enable-languages=`*lang* option, as well as some language-specific GCC configuration options, when building specific compilers. For GCC 3.x and earlier compilers, valid values for `lang` are `ada`, `c`, `c++`, `f77`, `java`, and `objc`. For GCC 4.x compilers, valid values for `lang` are `ada`, `c`, `c++`, `fortran95`, `java`, `objc`, and `obj-c++`. The new name for GCC's new Fortran compiler and the slight difference in naming between `objc` and `obj-c++` can be confusing if you've built GCC compilers before.

Notes for Building the GCC Fortran Compiler

As mentioned earlier in this chapter in the section titled "Verifying Software Requirements," you must have the GMP and MPFR libraries installed on your system or the configuration step will fail.

Notes for Building the GCC Java Compiler

If you are building the GCC Java compiler but either do not want to rebuild the runtime libraries or wish to rely on some other Java runtime libraries, you can specify the `--disable-libgcj` option. One common use of `--disable-libgcj` is when you want to use the gcj compiler with another Java compiler's runtime libraries. (See `https://www.redhat.com/magazine/012oct05/features/java/` and `http://lists.gnu.org/archive/html/classpath/2003-12/msg00120.html` for information about doing this and other Java runtimes available for Linux.) You must use the GNU Java runtime to use other bindings that explicitly depend on it, such as the Java-GNOME bindings for Java.

Notes for Building the GCC Objective-C Compilers

If you are compiling the Objective-C or Objective-C++ compilers, GCC 4.x provides support for a memory management policy implemented by the Boehm-Demers-Weiser conservative garbage collector. As mentioned in Table 11-2, this garbage collector is available in source form from `http://www.hpl.hp.com/personal/Hans_Boehm/gc/` and must be built and installed before configuring GCC or compiling its Objective-C or Objective-C++ compilers. Once the garbage collector is built and installed, you can configure GCC to use it through the `--enable-objc-gc` option to the configure command. Specifying this option and building the Objective-C or Objective-C++ compilers will build an additional runtime library, libobjc_gc.a, which has several enhancements to support the garbage collector and has a new name so as not to conflict with the name of the default objective-C runtime library (libobjc.a).

Compiling the Compilers

Now that you have all of the preliminaries out of the way, you are finally ready to build the compiler. In most situations, you only need to use one of two commands, `make bootstrap` or `make bootstrap-lean`. Briefly, these two `make` commands build the new compiler from scratch according to the configura-

tion you created with the configure script. My suggestion is to put this book aside for a moment, type `make bootstrap` to start the build process, and then finish reading this chapter while the build takes place.

Be clear, of course, that a full bootstrap is time consuming, even on fast machines. Building GCC from scratch can take from half an hour to several hours, depending on the speed of your processor, the amount of memory available, and the system load. The build process definitely benefits from a faster CPU and more RAM. This section of the chapter explains the build process in detail, or at least in enough detail to understand why it may take a while.

■ **Tip** To maximize the number of CPU cycles and the amount of RAM available to the build process, disable as many nonessential processes running on the build system as possible. Likewise, if you have sufficient administrative privileges on the build system and if the operating system supports it, you can experiment with increasing the priority of the build process. On Linux and Unix systems, you would do this using the `renice` command (as the root user) to modify the priority of the top-level shell running the build (the one executing `make bootstrap` or `make bootstrap-lean`) or by starting the build as the root user (not recommended, by the way) and using the `nice` command when you start the build.

Compilation Phases

The recommended and preferred method for building the compiler is using the `make bootstrap` command. The discussion that follows applies to a full bootstrap (executed by `make bootstrap`) but can also be applied to the `make bootstrap-lean` target unless noted otherwise. A full bootstrap build executes the following steps:

1. Build all host tools required to build the compiler(s), such as Texinfo, Bison, gcov, and gperf. (See the section "Verifying Software Requirements," earlier in this chapter.)
2. If the binary utilities for the compiler target (bfd, binutils, gas, gprof, ld, and opcodes) were moved or linked to the top-level GCC build directory prior to executing configure, build them.
3. Perform a three-stage build of the compiler itself. First, build the new compiler with the native compiler, which results in the stage 1 compiler. Next, build the new compiler, known as the stage 2 compiler, with the stage 1 compiler. Last, build the new compiler a third time (the stage 3 compiler) with the stage 2 compiler.
4. Compare the stage 2 and stage 3 compilers to each other. They should be identical; otherwise, the stage 2 compiler is probably flawed and, as a result, you should not install the stage 3 compiler.
5. Using the stage 3 compiler, compile the needed runtime libraries.

If you use `make bootstrap-lean` instead of `make bootstrap`, the object files from stages 1 and 2 will be deleted after they are no longer needed, minimizing disk space consumption.

During the build process, you will probably see a number of warning messages scroll by. Although disconcerting, they are warnings, not errors, and so you can safely disregard them. As a rule, these warnings are limited to certain files. You only need to pay attention to actual errors that cause the build process to fail. In other cases, certain commands executed during the build process will return a nonzero error code, which the make program recognizes as a failure but nevertheless ignores. Often as not, the "failure" is simply a missing file that is not relevant and like the warning messages mention, it is safe to ignore such failures unless they cause the build process to terminate abnormally.

If you have an SMP system (lucky you!), you can start a parallel build—a build that takes advantage of all the CPUs in your system—by executing the following command if your version of GNU make is older than 3.79:

```
$ MAKE="make -j 2" make bootstrap -j 2
```

The reason you have to use the `MAKE` environment variable is because versions of make older than 3.79 do not pass the `-j 2` option down to child make processes (so-called *submakes*). By setting the `MAKE` environment variable, which is inherited by submakes, you get the parallelism you want. If you have version 3.79 or newer of GNU make, a simple

```
$ make -j 2 bootstrap
```

will be sufficient. The numeric argument to `-j` should be no greater than the number of CPUs you have in your system—although making it greater will work, it will not result in any better performance.

Other Make Targets

As you might surmise, the GCC build process supports many more targets than `bootstrap` and `bootstrap-lean`. Table 11-3 lists the most useful make targets (*makeables*) other than `bootstrap` and `bootstrap-lean`.

Table 11-3. *GCC Makefile Targets*

Target	Description
`all`	Coordinates building all of the other targets, according to the values of host, build, and target you specify (if any) when you execute the configure script.
`bubblestrap`	Incrementally rebuilds each stage using any previously completed `bootstrap` (or `bubblestrap`) build stages. This can be useful if a previous stage of `make bootstrap` failed and you want to continue building with the last successfully completed build stage, rather than rebuilding everything.
`check`	Executes the test suite.
`clean`	Performs the `mostlyclean` step and also deletes all other files created by `make all`.
`cleanstrap`	Executes `make clean` followed by `make bootstrap`.
`compare`	Compares the results of stages 2 and 3, which should be the same.
`distclean`	Deletes the same files as `clean`, plus files generated by the configure script.
`dvi`	Generates the DVI-formatted documentation, which is suitable for printing.
`extraclean`	Executes `make clean` and also deletes temporary and backup files created during the build.
`generated-manpages`	Creates the generated manual pages, which are a subset of the complete Texinfo documentation.
`install`	Installs GCC into the directory or directories specified when you execute the configure script.
`maintainer-clean`	Deletes the same files as `make distclean` and also deletes all files generated from other files.
`mostlyclean`	Deletes the files created by the build process.

Table 11-3. *GCC Makefile Targets (Continued)*

Target	Description
`quickstrap`	Rebuilds the most recently completed stage, meaning you do not have to keep track of the stage you are on.
`restage`*N*	Executes `make unstage`*N* and then rebuilds stage *N* with the appropriate compiler flags, where *N* is `1`, `2`, `3`, or `4`.
`stage`*N*	Moves the files for each stage *N*, where *N* is `1`, `2`, `3`, or `4`, to the appropriate subdirectory in the build tree.
`uninstall`	Deletes the installed files (except for documentation). Note that `make uninstall` might not work properly.
`unstage`*N*	Reverses the actions of stage *N*, where *N* is `1`, `2`, `3`, or `4`.

Testing the Build

I strongly recommend you test your shiny new compiler before you install it, even though the GCC documentation describes running the test suite as an optional step. The primary reason I recommend testing the compiler is that it will be used to create the software you use every day. The time required to run the test suite seems like a small price to pay for making sure you do not have a flawed or broken compiler. In most cases, you will not encounter any serious errors, but in my opinion, it does not hurt to make sure. This section explains how to install and run the test suite and briefly describes how to interpret the results.

In order to run the GCC test suite, you need Tcl, Expect, and DejaGNU installed. Tcl is the Tool Command Language, and it is usually part of standard Linux or Unix systems that have a working development environment. On Windows-based systems, you probably do not have it installed unless you specifically installed it. Expect is a Tcl-based tool that automates interactive programs. It is rather less commonly installed than Tcl. See the section earlier in this chapter titled "Verifying Software Requirements" for information on downloading, building, and installing these packages.

To see if you have Tcl and Expect installed, execute the commands `which tcl` and `which expect`. The output should resemble the following:

```
$ which tcl
```

```
/usr/bin/tcl
```

```
$ which expect
```

```
/usr/bin/expect
```

If the output resembles the following lines, the package is not installed, or at least is not in your `PATH`:

```
$ which tcl
```

```
which: no tcl in (/bin:/usr/bin:/usr/local/bin)
```

Assuming that you have Tcl and Expect installed, the next step is to unpack, build, and install the test harness, which automates running the tests. A test harness is the term used to describe the framework in which tests are executed. For GCC, the test harness is DejaGNU, which can be downloaded from the GNU FTP site or one of its mirrors. If you download from the primary GNU FTP site, the URL is `ftp://ftp.gnu.org/gnu/dejagnu/dejagnu-VERSION.tar.gz`.

After you download the file, unpack and install DejaGNU. Using DejaGNU version 1.4.4 as an example, the sequence of commands for my crash test dummy system is as follows (I used the same `/home/wvh/src` directory I used for unpacking the GCC distribution):

```
$ tar -zxf dejagnu-1.4.4.tar.gz
$ cd dejagnu-1.4.4
$ ./configure
$ make
$ su
Password:
# make install
```

I recommend using the default installation directory, `/usr/local`, to limit the chances of something going wrong with the test process. The `make install` step requires root privileges, so use the `su` command to become the superuser. These commands unpack the source tarball, configure it for my combination of CPU and Linux version, build the test suite, and install it into its default location. "Building" the test harness is somewhat misleading, however, since DejaGNU is based on Expect, which in turn is based on Tcl. Tcl is an interpreted programming language that does not require compilation. All that the "build process" really does is set a number of variables that are used by DejaGNU. In fact, I ordinarily build and install DejaGNU when I unpack the GCC sources just to save a step later.

■ **Caution** Make sure you unpack the GCC test suite itself, as explained in the section titled "Installing the Source Code" earlier in this chapter. The test process will fail if the test suite has not been installed.

After installing DejaGNU, you might need to set the environment variables `DEJAGNULIBS` and `TCL_LIBRARY` as shown in the following code lines, modifying the pathnames to reflect your particular situation. If Tcl is in your `PATH` and if you use the default installation directory for DejaGNU, however, you will not need to set `TCL_LIBRARY` or `DEJAGNULIBS`.

```
$ export DEJAGNULIBS=/usr/local/share/dejagnu
$ export TCL_LIBRARY=/usr/lib/tcl8.3
```

Finally, make the top-level GCC build directory your current working directory and invoke the test suite using the `make` command shown in the following:

```
$ cd /home/wvh/src/gcc-obj
$ make -k check
```

If invoked as shown in this example, the testing process will validate as many of the compiler components as possible, thus exercising the C, C++, Objective C, and Fortran compilers and validating the C++ and Java runtime libraries. If you wish, you can run subsets of the test suite. For example, to test only the C compiler, execute the following commands:

```
$ cd /home/wvh/src/gcc-obj/gcc
$ make check-gcc
```

Similarly, to test only the C++ compiler, use the following two commands:

```
$ cd  /home/wvh/src/gcc-obj/gcc
$ make check-g++
```

In both of these examples, the directory `gcc` is the gcc subdirectory of the top-level GCC build directory. Like the build process, running the test suite can be quite time-consuming, and like the build process, the faster the host system's CPU, the faster the test suite will complete.

Interpreting the test results from a high level, that is, without getting into the details of the test results, is relatively simple. In each of the test suite subdirectories, the test process creates a number of files ending with `.log`, which contain detailed test results, and `.sum`, which summarize the test results. The summary files list every test that is run and a one-word code indicating the test result. Table 11-4 lists the possible codes and their meanings.

Table 11-4. *GCC Test Suite Result Codes*

Code	Description
ERROR	The test suite encountered an unexpected error.
FAIL	The test failed but was not expected to do so.
PASS	The test completed successfully and was expected to do so.
UNSUPPORTED	The corresponding test is not supported for the platform in question.
WARNING	The test suite encountered a potential problem.
XFAIL	The test failed and was expected to do so.
XPASS	The test completed successfully but was not expected to do so.

Some tests will result in unexpected failures, such as tests that deliberately torture the compiler's ability to handle extreme situations. Unfortunately, the test suite does not support a greater degree of control over this situation.

■ **Note** The test suite is currently in a state of flux because the GCC steering committee, with input from developers, is considering moving to a more robust and flexible test harness, so the situation may have changed by the time you read this book.

You can also submit your test results to the GCC project by using the test_summary script, which bundles up the summary files and uses the mail program to send the test results to a server where additional processing takes place. To use the script, execute the following commands:

```
$ cd /home/wvh/src/gcc-VERSION
$ contrib/test_summary [-p extra_info.txt] -m gcc-testresults@gcc.gnu.org | /bin/sh
```

The file extra_info.txt can contain any information you think pertinent to the test results. It is not required, however, so you can omit this element of the command if you wish.

Installing GCC

At last, you are ready to install GCC. Your options are limited here, because the installation paths and other installation-related details were largely determined when you ran the configure script. The most commonly used command to install your shiny new compiler is `make install` and it must be executed by the root user unless you are installing it into nonsystem directories (that is, directories to which you have write access):

```
$ su -
Password:
# cd /home/wvh/src/gcc-obj
# make install
# ldconfig -v
```

The `ldconfig` command updates the dynamic linker's cache file, allowing it to pick up the new shared libraries (libgcc, and possibly the runtime libraries for C++ and Java). It is not strictly necessary, but it doesn't hurt, either.

That's it! After considerable time, minimal effort on your part, and a Herculean workout of your CPU, you have a brand new compiler installed. If you are curious about why you have gone to all the trouble of installing a new compiler, reread the section in the introduction to this book titled "Why the New Edition?" Sometimes the latest and greatest is indeed the best.

CHAPTER 12

Building and Installing Glibc

The GNU C library, popularly known as Glibc, is the unseen force that makes GCC, most C language applications compiled with GCC on Linux systems, and all GNU/Linux systems work. Any code library provides a set of functions that simplify writing certain types of applications. In the case of the GNU C library, the applications facilitated by Glibc are any C programming language applications. The functions provided by Glibc range from functions as fundamental to C as `printf()` to Portable Operating System Interface for Computer Environments (POSIX) functions for opening low-level network connections (more about the latter later). The C programming language actually has a relatively small number of keywords; most of what people think of as C is actually standard C library functions that are implemented in Glibc.

Though Glibc is quite stable (aptly demonstrated by the fact that most Linux applications depend on it), it is indeed software. You may want to upgrade to a newer version for a variety of reasons, the most common of which are to resolve some problem that you've discovered, to take advantage of improvements in how existing functions are implemented in a newer version of Glibc, or to take advantage of new functions provided in the latest and greatest versions.

Rebuilding and changing the library on which almost every GNU/Linux application depends can be an intimidating thought. Luckily, it is not as problematic or complex as you might think. This chapter explains how to obtain, build, and install newer versions of the GNU C library, discusses problems that you might encounter, and also explains how to work around those problems in order to get your system(s) up and running with newer or alternate versions of the GNU C library that may be present by default on the system(s) that you are using.

Note The most important requirement for upgrading Glibc on your system is having sufficient disk space to download and build the source code, back up files that you want to preserve in case you encounter upgrade problems, and install the new version of Glibc. You should make sure that your system has approximately 200MB of free space in order to build and install Glibc.

What Is in Glibc?

Because most of the fundamental Un*x, Linux, HURD, and BSD applications are written in the C programming language, every Unix-like operating system needs a C library to provide the basic capabilities required in C applications and to provide the system calls that enable C applications to interface with the operating system. Glibc is the one true C library in the GNU system, and in most newer systems with the Linux kernel.

The contents of interfaces provided as part of Glibc have evolved over time and reflect the history of Unix and relevant standards. Glibc provides support for the following standards and major Un*x variants:

- *Berkeley Standard Distribution (BSD)*: No one with any history on Un*x systems could be unaware of the enhancements to Un*x that were provided by the Berkeley Standard Distribution. NetBSD, FreeBSD, and even Apple's Mac OS X still carry the flag of many of the networking, I/O, and usability improvements made to Un*x by the University of California at Berkeley and other academic institutions such as Carnegie Mellon University, and long-lived BSD-based Un*x implementations such as Sun Microsystems SunOS. Glibc supports many of the capabilities found in the 4.2 BSD, 4.3 BSD, and 4.4 BSD Unix systems, and in SunOS. This heightens code compatibility with 4.4 BSD and later SunOS 4.X distributions, which themselves support almost all of the capabilities of the ISO C and POSIX standards.
- *ISO C*: This is the C programming language standard adopted by the American National Standards Institute (ANSI) in the "American National Standard X3.159-1989—ANSI C" document, and later by the International Standardization Organization (ISO) in its "ISO/IEC 9899:1990, Programming Languages—C" document. This is colloquially referred to as the *ISO C standard* throughout the Glibc documentation.

■ **Note** The header files and functions provided by Glibc are a superset of those specified in the ISO C standard. If you need to write applications that strictly follow the ISO C standard, you must use the `-ansi` option when compiling programs with GCC. This will identify any non-ANSI constructs that you might have used accidentally.

- *POSIX:* This is the Portable Operating System Interface for Computer Environments document (ISO/IEC 9945-1:1996), later adopted by ANSI and the IEEE (Institute of Electrical and Electronics Engineers) as ANSI/IEEE Std 1003. The POSIX standard has its roots in Unix systems and was designed as a standard that would facilitate developing applications that could be compiled across all compliant Un*x systems. The POSIX standard is a superset of the ISO C standard, adding new functions and extending existing functions in the ISO C standard. If you need to sling applicable acronyms, the Glibc manual states that Glibc is compliant with POSIX, POSIX.1, IEEE Std 1003.1 (and IEEE Std 1003.2, Draft 11), ISO/IEC 9945-1, POSIX.2, and IEEE Std 1003.2. Glibc also implements some of the functionality required in the "POSIX Shell and Utilities" standard (a.k.a. POSIX.2 or ISO/IEC 9945-2:1993).
- *SYSV Unix:* Un*x history began at AT&T Bell Laboratories. Glibc supports the majority of the capabilities specified in the AT&T "System V Interface Description" (SVID) document, which is a superset of the POSIX standard mentioned earlier.
- *XPG:* The "X/Open Portability Guide," published by the X/Open Company, Ltd., is a general Un*x standard. This document specifies the requirements for systems that are intended to be conformant Unix systems. Glibc complies with the "X/Open Portability Guide, Issue 4.2," and supports all of the X/Open System Interface (XSI) and X/Open Unix extensions. This should not be a big surprise, since the majority of these are derived from enhancements to Unix made on System V or BSD Unix, and Glibc is compliant with those.

Today's Glibc has come a long way from the statically linked version 1.x Glibc of the 1980s. Today, Glibc is a powerful set of shared libraries that is used on hundreds of thousands of computer systems all over the world. Like GCC, Glibc is a living testimonial to the power of open source software and the insight and philanthropy of its designers and contributors.

■ **Note** Glibc isn't the only C library that you can use with GCC's C compiler. Several alternate C libraries are available and work quite well with GCC. Primarily developed for use in embedded applications where storage space and memory are at a premium, alternate C libraries such as dietlibc, klibc, Newlib, and uClibc support various subsets of C functions. See Chapter 13 for information on building and using these C libraries with GCC's C compiler.

Why Build Glibc from Source?

Like any piece of open source software, particularly one with hundreds of thousands of users, inadvertent quality assurance personnel, and potential contributors, Glibc is continually being enhanced and improved. Even aside from bug fixes (this is software, after all) and performance improvements, general enhancement and improved organization of Glibc is a continuous process.

The latest version of Glibc available at the time of this writing was Glibc 2.4, which was released in March of 2006. However, the change from Glibc 2.3.x to 2.4 is what is known as a major version change, in which significant changes to the C library can affect backward compatibility. When major version number changes occur, it is suggested that compilers be rebuilt using the new Glibc version, and that all object code on the system be recompiled and relinked using the new Glibc version. Similarly, some of the tests in the Glibc test suite will only function correctly against Linux kernels version 2.6.16 or better, which most machines did not provide out of the box at the time that this book was written. This chapter therefore focuses on upgrading your system's Glibc to 2.3.6, which was the current release of the Glibc 2.3.x family as of the time that this book was written. The instructions in this chapter for upgrading to 2.3.6 should also work for later versions of the Glibc 2.3.x family. For details on changes to Glibc 2.4 and suggestions for building and upgrading your system to that version of the C library, see the section at the end of this chapter titled "Moving to Glibc 2.4."

■ **Caution** Systems with Glibc 2.3.x installed should be able to successfully execute programs compiled under any newer 2.x releases of Glibc. Remember that compatibility only works in one direction—programs compiled on a system running Glibc 2.3.x or later will not run correctly (or at all) on systems with earlier versions of Glibc. Luckily, you have to go pretty far back in Linux distribution time (SUSE 8.x, Red Hat 8, Mandrake 9, Yellow Dog Linux 2.3, etc.) to find a distribution that predates Glibc 2.3, so you're probably in good shape.

As stated in its release announcement, Glibc 2.3.6 is actively supported on the following platforms and system types:

- *-*-gnu - GNU HURD
- alpha*-*-linux-gnu - Linux-2.x on DEC Alpha
- arm-*-none - ARM stand-alone systems
- arm-*-linux - Linux-2.x on ARM
- arm-*-linuxaout - Linux-2.x on ARM using a.out binaries
- i[3456]86-*-linux-gnu - Linux-2.x on Intel
- ia64-*-linux-gnu - Linux-2.x on IA-64
- m68k-*-linux-gnu - Linux-2.x on Motorola 680x0
- mips*-*-linux-gnu - Linux-2.x on MIPS
- powerpc-*-linux-gnu - Linux and MkLinux on PowerPC systems

- powerpc64-*-linux-gnu - Linux-2.4.19+ on 64-bit PowerPC systems
- s390-*-linux-gnu - Linux-2.x on IBM S/390
- s390x-*-linux-gnu - Linux-2.4+ on IBM S/390 64-bit
- sh-*-linux-gnu - Linux-2.x on Super Hitachi
- sparc-*-linux-gnu - Linux-2.x on SPARC
- sparc64-*-linux-gnu - Linux-2.x on UltraSPARC 64-bit
- x86-64-*-linux-gnu - Linux-2.4+ on x86-64

Platform names in this list use standard regular expression syntax—for example, * means any matching string in a specific field, and [abc] means *any of a, b, or c*.

For software developers, just staying ahead of (or on top of) the curve is a good argument for upgrading your system to newer versions of Glibc. While the basic set of functions used in C programming is not really going to change radically, implementations of those functions can. Making sure that your software will continue to compile and work correctly on new or upcoming releases of Glibc is a good thing.

Much of the work being done in Glibc nowadays consists of internal enhancements that reflect the increasing maturity of the Linux code base and the software development community in general. The most notable example of this is increased support for and emphasis on internationalization. A few years ago, i18n (the commonly used abbreviation for the word *internationalization*) was something that people would get to one of these days. Nowadays, the computing (and Linux) communities are truly international, and well-crafted applications provide intrinsic support for different languages and character sets in messages and graphical displays.

Potential Problems in Upgrading Glibc

Like any shared library, Glibc follows the naming and compatibility conventions discussed in Chapter 8. As a quick recap, the major, minor, and release version numbers of a shared library (*major.minor.release*) are updated based on the type of changes made between different versions of the shared library. When a new version of a shared library is released in order to fix bugs in a previous version, only the release number is incremented. When new functions are added to a shared library but existing functions in the library still provide the same interfaces, the minor version number is incremented and the release number is reset. When interfaces in functions in a shared library change in such a way that existing applications cannot transparently call those functions, the major number is updated and the minor and release numbers are reset. Applications that depend on specific interfaces to functions can then still load the shared library with the correct major and minor number at runtime, while applications compiled against a new version of the same shared library (featuring different parameters or a different parameter sequence) to existing functions can link against the version of the shared library with the new major number.

In a nutshell (sorry O'Reilly guys!), this means that it should be trivial to upgrade a system to a new version of Glibc with a higher release or a minor number. You can expect to encounter problems when upgrading across major numbers, because all of your existing applications will have to be relinked with the new Glibc: the basic symbolic links to the "well-known" version of Glibc (/lib/libc.so.6), the shared Linux loader library (/lib/ld-linux.so.2), and the POSIX threads library for your architecture (/lib/arch/libpthread.so.0, such as /lib/i686/libpthread.so.0 on i686 systems) will all have changed with the major version number.

You need not be paranoid when upgrading Glibc on your system, but you should be careful and aware of potential problems. The section "Troubleshooting Glibc Installation Problems" later in this chapter explains some problems that you may encounter and provides solutions and workarounds. Nowadays, upgrading Glibc is nowhere nearly as problematic as it was in earlier, more primeval

Linux days, such as when Linux systems converted from using libc version 5 to version 6, or when the executable binary format of all Linux applications changed from a.out to ELF. For one, those were the bad old days—life is simpler and better now.

Identifying Which Glibc a System Is Using

If you are considering upgrading to a newer version of Glibc, it is important to be able to determine what version of Glibc your system is actually using. The easiest way to do this is simply to list the main Glibc library on your system. The name of the primary C library on your system is a symbolic link to the version of the C library that is actually being used on your system. A symbolic link is used rather than an explicit filename because, although every C language program that uses shared libraries has to know the name of the primary C library, using a single well-known name ensures portability of executables that use shared libraries across multiple systems.

The primary Glibc shared library used by C programs on modern systems that use GCC is /lib/libc.so.6. Listing this in long format results in something like the following:

```
$ ls -al /lib/libc.so.6
```

```
lrwxrwxrwx  1 root  root   14 Jan 14 17:08 /lib/libc.so.6 -> libc-2.3.5.so
```

The system on which this command was executed is running Glibc version 2.3.5. You can confirm this by listing the other critical shared object filename on your system, which is the name of the Linux load library, as in the following example:

```
$ ls -al /lib/ld-linux.so.2
```

```
lrwxrwxrwx  1 root  root   12 Jan 14 17:08 /lib/ld-linux.so.2 -> ld-2.3.5.so
```

As an alternative to listing symlinks, you can either get verbose output by executing the C library itself (as explained in the next section), or write a small program that executes the specific Glibc `gnu_get_libc_version()` function that is provided to display Glibc version information. Though this does not provide any information beyond what you can infer from the names of the symbolic links, it is more empirical because it queries the library itself rather than installation details. The following is a small program named glibc_version.c that displays Glibc version information:

```
#include <stdio.h>
#include <gnu/libc-version.h>
int main (void) {
    puts (gnu_get_libc_version ());
    return 0;
}
```

Compiling and running this on a sample system produces output like the following:

```
$ gcc glibc_version.c -o glibc_version
$ ./glibc_version
```

```
2.3.5
```

Simply listing symbolic links or querying the library itself provides simple, high-level information about the version of Glibc that your system is running but does not actually tell you what is in it. As discussed in the section "Glibc Add-Ons," Glibc is traditionally compiled with a number of external capabilities, known as *extensions* or *add-ons,* that you retrieve separately and integrate into your Glibc source directory before you begin building Glibc. The next section explains how to get more detailed information about the capabilities provided by the version of Glibc that is installed on your system.

Getting More Details About Glibc Versions

Glibc provides a tremendous amount of functionality in a single library. The sets of functions that it provides by default can be augmented by incorporating add-ons, which are library source code modules that you must obtain separately and integrate into your Glibc source directory before you begin building Glibc. If you are planning to upgrade Glibc on your system(s), knowing the capabilities provided by the version of Glibc that your system is currently running is fairly important. This information enables you to ensure that the updated version of Glibc you build and install on your system provides the capabilities that the applications currently on your system require. As I'll discuss in the section "Downloading and Installing Source Code," the Glibc configuration process actively warns you if some basic add-ons are not present in the version of Glibc that you are configuring.

The easiest way to get detailed information about the version of Glibc that you are running on a particular system is to simply execute the main Glibc shared library (or its symbolic link) from the command line. This provides a wealth of information about the version of Glibc that you are running and also explains the add-ons/extensions that are integrated into that library. The following is some sample output from a SUSE 10.0 system:

```
$ /lib/libc.so.6
```

```
GNU C Library stable release version 2.3.5 (20050802), by Roland McGrath et al.
Copyright (C) 2005 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Configured for i686-suse-linux.
Compiled by GNU CC version 4.0.2 20050901 (prerelease) (SUSE Linux).
Compiled on a Linux 2.6.12 system on 2005-09-09.
Available extensions:
        GNU libio by Per Bothner
        crypt add-on version 2.1 by Michael Glad and others
        linuxthreads-0.10 by Xavier Leroy
        GNU Libidn by Simon Josefsson
        NoVersion patch for broken glibc 2.0 binaries
        BIND-8.2.3-T5B
        libthread_db work sponsored by Alpha Processor Inc
        NIS(YP)/NIS+ NSS modules 0.19 by Thorsten Kukuk
Thread-local storage support included.
For bug reporting instructions, please see:
<http://www.gnu.org/software/libc/bugs.html>.
```

This output shows that the system is indeed running Glibc version 2.3.5 and that it was compiled with a variety of extensions. How standard are these extensions? The following is sample output from a version of Glibc 2.3.6 that I built and installed for one of my systems:

```
$ /lib/libc.so.6
```

```
GNU C Library stable release version 2.3.6, by Roland McGrath et al.
Copyright (C) 2005 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 4.0.2 20050901 (prerelease) (SUSE Linux).
Compiled on a Linux 2.6.13-15.8-default system on 2006-03-21.
Available extensions:
        GNU libio by Per Bothner
        crypt add-on version 2.1 by Michael Glad and others
        GNU Libidn by Simon Josefsson
        Native POSIX Threads Library by Ulrich Drepper et al
        BIND-8.2.3-T5B
        NIS(YP)/NIS+ NSS modules 0.19 by Thorsten Kukuk
Thread-local storage support included.
For bug reporting instructions, please see:
<http://www.gnu.org/software/libc/bugs.html>.
```

Aside from the sequence in which the add-ons are listed, the only difference between these two is the difference in libraries associated with threading, which is discussed in more detail in the next section.

Glibc Add-Ons

The previous sections introduced Glibc add-ons, which provide separate, external functionality that you can integrate into a Glibc build by uncompressing them in your Glibc source directory and then configuring Glibc using the `--enable-add-ons` command-line option. As of Glibc 2.3.6 and greater, the only useful add-on is the libidn library for handling internationalized domain names.

In earlier 2.3.x releases of Glibc, other add-on packages that were suggested for use when building Glibc were the following:

- A POSIX-compliant thread library, glibc-linuxthreads, was used on almost all supported GNU/Linux systems. The `libthreads` module is so standard that its numbering conventions follow those of Glibc itself. When downloading Glibc in order to compile it for a GNU/Linux system, you should also obtain the version of the archive for the glibc-linuxthreads library that corresponds to the Glibc version that you will be building. For Glibc versions 2.3.6 and later, threading support is provided by the built-in Native POSIX Threading Library (NPTL).
- The Berkeley DB NSS (Name Service Switch) module, previously integrated into earlier versions of the GNU C library, was broken out into a separate `nss-db` add-on. This add-on enables you to store host information in Berkeley DB format and, thus, to use `db` as a meaningful entry in the name service configuration file /etc/nsswitch.conf. The `nss-db` add-on provides support for Berkeley DB releases 2.x and 3.x. If you are still dependent on Berkeley DB 1.x, you can obtain the source code for a compatible version of the library from Sleepycat Software (`http://www.sleepycat.com`), compile it separately as a shared library, and install it in the system library directory (`/lib`).
- The `nss-lwres` module supports the lightweight resolver in Berkeley Internet Name Daemon (BIND) 9 on systems that use Glibc. The lightweight resolver is a combination of library functions and a daemon process that reduces name service lookup overhead when compared to full-blown DNS. For more information about the lightweight resolver in BIND 9, see Section 5 of the BIND 9 documentation. A sample link to this documentation is `http://www.csd.uwo.ca/staff/magi/doc/bind9/Bv9ARM.ch05.html`.

If you have built even earlier versions of Glibc, you may notice that some of the other add-ons you built with older versions of Glibc are no longer used or required. The most common of these, and the reasons why they are no longer used, are the following:

- `glibc-compat`: Glibc 2.1 introduced some fundamental (and incompatible) changes in the Glibc IO library, libio. The `glibc-compat` add-on provides a libcompat.a library, older `nss` modules, and a few other files that make it possible to do development with old static libraries on a system running Glibc 2.1. Unless you are maintaining an older system or need to be able to continue to run Glibc 2.0 binaries on a modern Linux system, you do not need this add-on. If you do need it, it can be challenging to find, but is certainly still available on the Internet somewhere. Is not everything?
- `glibc-crypt`: Glibc 2.2 and later versions no longer require a separate cryptology add-on. In a rare burst of cluefulness, U.S. export regulations were relaxed so that the DES cryptographic functions could be incorporated into the base GNU C library distribution without sending anyone to jail. If anyone knows who was responsible for this breakthrough, perhaps you could have them take a look at similar foolishness such as the DMCA (Digital Millennium Copyright Act) and the one-click patent!
- `localedata`: Glibc 2.1 and later versions no longer require a separate module for easily producing internationalized code. Maybe this is one world, after all.

If you are upgrading an existing version of Glibc to a version that requires the `glibc-crypt` or `localedata` add-ons, you can still obtain them from the primary Glibc download site, `http://ftp.gnu.org/gnu/glibc/`, or any of its mirrors. The latest versions of these add-ons available at the time of this writing were the following:

- `glibc-crypt-2.1.tar.gz` (40K)
- `glibc-localedata-2.0.6.tar.gz` (511K)

Now that you know what is in your existing Glibc, the following section provides an overview of the process required to download, configure, build, and install a different Glibc than the one that is already running on your system.

Previewing the Build Process

As explained throughout the remainder of this chapter, the process of building and installing a new version of Glibc on your system is actually fairly simple, thanks to fine tools from the Free Software Foundation such as Autoconf and Automake, which produce a configure script for Glibc that provides a flexible, automatic configuration interface for Glibc. The general process of building Glibc on your system is as follows.

■ **Note** Before beginning the process of replacing or updating Glibc on your system, make sure that you create a rescue disk so that you can still boot your system if something goes wrong during the Glibc installation/update process. For information about building a rescue disk, see the section later in this chapter titled "Using a Rescue Disk," or online sources of information such as `http://www.linuxplanet.com/linuxplanet/reports/4294` and `http://www.hrlug.org/rescuedisk.html`.

---

1. Download and unpack the version of Glibc that you want to upgrade your system to. The version used in the examples in this book is Glibc 2.3.6.
2. Configure the Glibc source code directory for your system by running the configure script with the options that are appropriate for your system.
3. Compile the Glibc source code using the `env LANGUAGE=C LC_ALL=C make` command. This compiles Glibc with standard default locale settings (in the `LANGUAGE` and `LC_ALL` environment variables). The `LANGUAGE` environment variable specifies the language to use in error messages. The `LC_ALL` environment variable specifies the default font set for displaying those messages.
4. Assuming that Glibc compiled correctly, verify the compiled Glibc using the command `env LANGUAGE=C LC_ALL=C make check`.
5. If you have configured Glibc as a replacement for your system's existing C library and want to upgrade your system, you should perform the remainder of the steps in this procedure. If you are installing it as an alternate to your system's C library, you are done.
6. Assuming that all of the Glibc tests pass successfully, shut your system down to single-user mode (just to be on the safe side—this is mandatory if other users are using the system), and install the new version of Glibc using the `env LANGUAGE=C LC_ALL=C make install` command.

■ **Note** On most Linux systems, you can shut your system down to single-user mode by executing the command `telinit 1`.

---

7. Execute the `ldconfig` command to refresh your system's library cache.

■ **Note** Do not panic if you experience problems at this point. See the section "Troubleshooting Glibc Installation Problems" for information about resolving Glibc upgrade problems. A truly nice feature of installing newer versions of Glibc is that the new libraries are installed but the existing libraries are left in place.

---

8. Execute a few of your favorite commands to ensure that everything is still working correctly with the new C library.
9. Compile a sample application to ensure that GCC still works correctly.
10. Reboot.

That is all there is to it! Depending on your system's configuration, the process might require relinking or rebuilding your system's library cache, but upgrading Glibc is nowhere nearly as complex as it was during some of the major Linux library upheavals, such as when Linux systems converted from using libc version 5 to version 6, or when the executable binary format of all Linux applications changed from a.out to ELF. You may also find that you need to relink or update some of the applications that you may have compiled for your system if they have dependencies on misfeatures of previous functions in the Glibc library or related libraries.

Subsequent sections discuss these steps in more detail, walking you through all of the nuances of each stage. Sections where you might encounter a problem contain sidebars discussing these potential problems and provide workarounds.

Recommended Tools for Building Glibc

Each Glibc source code distribution provides a text file called INSTALL, which contains up-to-date information about building and installing the version of Glibc that it accompanies. This section summarizes the information about suggested, and in some cases required, versions of development utilities that you should have on your system in order to correctly build and install the version of Glibc that was current when this book was written, Glibc 2.3.6. Rather than trying to verify the version of each of these utilities that is installed on your system, I suggest that you consult this section only if you encounter a problem compiling Glibc.

Building and installing Glibc requires that up-to-date versions of the following utilities are installed on your system.

GNU awk 3.0 (or other POSIX awk): The awk utility, a feature of Un*x systems since it was written by Alfred Aho, Peter Weinberger, and Brian Kernighan at Bell Labs in the 1970s (its name comes from the initials of their last names), is a pattern-matching and data-reformatting language that is used when building Glibc (and most other GNU utilities). GNU awk is compliant with the awk specification in the "POSIX Command Language and Utilities" specification, but you can actually use any POSIX-compliant awk implementation to build Glibc. In most cases, I suggest that you install GNU awk just to be safe—after all, you can't beat the price. The source for the latest version of GNU awk is its home page, `http://www.gnu.org/software/gawk/gawk.html`. You can determine the version of awk that is installed on your system by executing the command `awk –version`.

GNU binutils 2.13 or later: The GNU binutils package includes utilities such as the GNU assembler (as) and the GNU linker/loader (ld). You must use these utilities with Glibc—only the GNU assembler and linker/loader have the features required to successfully compile and link applications that use Glibc.

GNU make 3.79 or newer: Like most GNU utilities and libraries, Glibc makes heavy use of the features provided by GNU make in order to simplify configuring, building, managing, and installing Glibc. GNU make is the standard make nowadays anyway. But if you experience problems or error messages after executing the `make` command to build Glibc, execute the `make -v` command to determine if you are running GNU make and what version it is. GNU make is truly a good thing. You can get the source code for the latest version from its home page, `http://www.gnu.org/software/make/make.html`. Prepackaged binary versions for most platforms are also available. To find prepackaged Linux binaries, try `http://www.rpmfind.net`. Solaris versions are available at the impressively useful `http://www.sunfreeware.com` Web site.

GCC 4 or newer: If you are building the version of Glibc used as an example throughout this chapter (2.3.6) or a newer version, you will want to use the latest and greatest version of GCC. Version 2.3 and greater of Glibc require a minimum of GCC 3.2 and will build fine with GCC 4.x or better. I won't insult you by describing GCC in more detail here—you own this book, right? See Chapter 11 for more information about downloading and building the latest version of GCC from source.

GNU sed 3.02 or newer: Like awk, sed (stream editor) is an ancient and ubiquitous Un*x pattern-matching and transformation utility of which an enhanced GNU version is freely available. I strongly suggest that you get the latest version of GNU sed if you are building Glibc. GNU sed is available from its home page at `http://www.gnu.org/software/sed/sed.html`. Though most versions of sed should be sufficient to build Glibc, some of the tests for validating Glibc will only work correctly if you use GNU sed. You can determine the version of awk that is installed on your system by executing the command `sed –version`.

GNU Texinfo 3.12f or better: The GNU Texinfo utility is the basis of all of the online help for Glibc and most GNU utilities. The classic online reference format used by all other Un*x/Linux utilities is not the primary help mechanism for GNU utilities—up-to-date help for Glibc (as for GCC) is compiled and installed in Info format instead. Developing a "new" utility for encoding and displaying help for command-line utilities may or may not have been a good idea, but regardless of how you feel, you will need an appropriate version of Texinfo to build and install the online documentation for Glibc. See Appendix C for more information about Texinfo. The source for the latest version of Texinfo is its home page at `http://www.gnu.org/software/texinfo/texinfo.html`.

Perl 5 or better: Perl, the magnificent programming language, can opener, and floor wax, written by Larry Wall (and others), is not required in order to build Glibc, but is used when testing Glibc. It is hard to conceive of a modern Un*x system that does not have Perl installed. You can obtain the sources for the latest version of Perl and binary distributions of Perl from the Comprehensive Perl Archive Network site at `http://www.cpan.org/`. If you are building Glibc for a relatively recent Linux system, it is highly likely that most of these utilities support the `--version command-line` option to display version information. If you need to upgrade any of them, the process for downloading, building, and installing them is the same as that for GCC or Glibc.

Note You can compile applications that use Glibc with any modern compiler that you like, but if you are using GCC, I strongly suggest using the latest version of GCC whenever possible. Older versions of GCC have various bugs that may be triggered by various libraries in newer versions of Glibc.

Updating GNU Utilities

If you are building and installing your own versions of these utilities, you should consider using an appropriate version of the `--prefix` command-line option when configuring them, so that the versions you are building will overwrite any older versions that are available on your system. To determine the correct value for `--prefix`, you can first use the `which` command to determine where the existing utility is installed on your system, and then specify an appropriate argument to `--prefix`. For example, determining where awk is installed on your system would look something like the following:

```
$ which awk
```

```
/usr/bin/awk
```

In this case, you would then specify `--prefix=/usr` as an argument to the GNU awk configure script in order to configure your awk source code directory such that executing the `make install` command would overwrite the installed version of awk. You could always simply build and install awk with its default values, which would install it as /usr/local/bin/awk, but you would then have to remember to set your path correctly (and the library path, for some of the other utilities) when building Glibc to ensure that the build process uses the "right" version of awk.

Downloading and Installing Source Code

This section explains how to download the basic archive files required for building and installing Glibc, and how to extract the contents of those archives.

Downloading the Source Code

The source code for all 2.x versions of Glibc and related add-ons is available at the primary Glibc download site (`ftp://ftp.gnu.org/pub/gnu/glibc`) or one of its mirrors.

You can download the sources and related add-ons using a browser or your favorite FTP client. Glibc source code archives are available there in gzip and bzip2 formats. Add-ons there are archived in gzip format. Extracting the contents of archive files in both of the gzip and bzip2 formats is explained later in the section "Installing Source Code Archives."

Unless you have a specific reason to not do so, you should always download and build the latest version of Glibc and related add-ons that are available. At the time that this book was written, the latest version of Glibc was 2.3.6, the source for which is contained in either of the following two archive files:

- glibc-2.3.6.tar.bz2
- glibc-2.3.6.tar.gz

The `.bz2` extension indicates that the first file is compressed using the bzip2 utility, which offers greater compression than the standard gzip compression utility, and thus is gaining in popularity. Extracting the contents of archives in both of these formats is explained in the next section.

As explained in the section "Glibc Add-Ons," you will also want to download any Glibc add-ons that you wish to install with Glibc. For the 2.3.6 and better versions of Glibc, there are no longer any critical add-ons, since thread support and NSS (Name Service Switch) name resolution are now built into Glibc. However, you will probably want to download the version of libidn, the library that handles internationalized domain names, which is associated with the version of Glibc that you are building.

Installing Source Code Archives

Once you have downloaded the source code for Glibc and any add-ons that you want to install, unpacking the archives is easy. All Glibc archives are compressed archives that are written in tar (tape archiver) format. The tar utility is a classic Un*x application that is still found on all Un*x systems, but has been reimplemented and extended (as always) by our friends at the Free Software Foundation.

■ **Note** You must use the appropriate commands explained in this section to extract the contents of both the Glibc source code archive and the archives of any add-ons that you downloaded before you proceed to subsequent sections of this chapter.

The arguments to the commands used to unpack archive files differ slightly depending on whether you are using GNU tar and whether you are extracting the contents of an archive that was compressed with gzip (and therefore has a `.gz` extension) or bzip2 (which therefore has a `.bz2` extension). The gzip compression/decompression scheme is described in RFC 1952, and is a Lempel-Ziv coding (LZ77) with a 32-bit CRC. The bzip and bzip2 compression/decompression scheme is based on the Burrows-Wheeler block-sorting-based lossless compression algorithm, which offers superior compression to gzip but is not as widely used.

If you are using a Linux system or are working under Cygwin, you are using GNU tar. If you are using a Solaris or other Unix system, you are already making a big commitment to GNU software by installing GCC and Glibc. Why not install GNU tar as well? You can obtain GNU tar from its home page at `http://www.gnu.org/software/tar/tar.html`.

If you are using GNU tar, you can unpack a gzipped archive into the current directory, and execute a command such as the following:

```
$ tar xzvf <archive-file>
```

The `x` option tells GNU tar to extract the contents of the specified archive file. The `z` option tells GNU tar to process the archive on the fly using the gzip compression scheme (in this case decompressing the archive file). The `v` option specifies verbose output, which lists files and directories as they are extracted from the archive. The `f` option identifies the following argument as the name of an archive file.

If you are not using GNU tar, you can extract the contents of any gzipped archive using the gzip program in tandem with the tar program. (If gzip is not installed on your system, get it now. You can download the source code for gzip or precompiled binaries from its home page at `http://www.gzip.org`.)

To extract the contents of a gzipped archive if you are not using GNU tar, execute a command such as the following:

```
$ gzip -cd <archive-file> | tar xvf -
```

The `c` argument to gzip directs output to the standard output stream, and the `d` option specifies decompression. The arguments to tar have the same meanings as the same arguments to GNU tar. The `-` tells the `tar` command to read from standard input rather than from an archive file.

To extract the contents of an archive file that is compressed using bzip2, execute a command such as the following:

```
$ tar xjvf <archive-file>
```

The `x` option tells GNU tar to extract the contents of the specified archive file. The `j` option tells GNU tar to process the archive on the fly using the bzip2 compression scheme (in this case decompressing the archive file). The `v` option specifies verbose output, which lists files and directories as they are extracted from the archive. The `f` option identifies the following argument as the name of an archive file.

If you are not using GNU tar, you can extract the contents of any bzipped archive using the bzip2 program in tandem with the tar program. To extract the contents of an archive that is compressed with bzip2 if you are not using GNU tar, execute a command such as the following:

```
$ bzip2 -cd <archive-file> | tar xvf -
```

The `c` argument to bzip2 directs output to the standard output stream, and the `d` option specifies decompression. The arguments to tar have the same meanings as the same arguments to GNU tar. The `-` tells the `tar` command to read from standard input rather than from an archive file.

Once de-archived, the source code for Glibc requires approximately 120MB.

THE LINUX STANDARD BASE

The Linux Standard Base is an up-and-coming standard for the applications that one should be able to find on Linux systems, including where they are located and the command-line arguments that they should provide. The goal of the Linux Standard Base is to provide a single standard that all Linux distributions can move toward.

One of the items discussed in the Linux Standard Base is the tar program and its options. To avoid an increasing number of options, one for each new compression scheme, the Linux Standard Base specifies that you should define the compression/decompression method as a separate argument, preceded with two dashes, rather than using options such as `j` (bzip2) or `z` (gzip). Equivalent commands to the previous two on systems that are compiled with the Linux Standard Base are the following:

```
$ tar xvf <archive-file> --gzip
$ tar xvf <archive-file> --bzip2
```

This is the command-line structure of the future for tar, separating the actions of external utilities from those specific to tar itself, and simplifying the integration of new, even faster compression mechanisms as they are developed.

Integrating Add-Ons into the Glibc Source Code Directory

Before configuring the Glibc source code and actually building Glibc, the last phase, so that Glibc can be built successfully, is to integrate the source code for any add-ons that you want to use. For more information on Glibc add-ons, see the section "Glibc Add-Ons."

As mentioned earlier, the only add-on that is really used with Glibc nowadays is the libidn library for handling internationalized domain names. When you extract this into the directory where you download it, the library has a name of the form glibc-libidn-*version*, where *version* should be the same as the version of Glibc that you are building.

To integrate the source code for the `libidn` add-on with your Glibc source directory, you simply move its source code into the Glibc source directory, giving it the generic name of `libidn` rather than retaining its version number, as in the following example:

```
$ pwd
```

```
/usr/local/src
```

```
$ ls
```

```
glibc-2.3.6   glibc-libidn-2.3.6
```

```
$ mv glibc-libidn-2.3.6 glibc-2.3.2/libidn
$ ls
```

```
glibc-2.3.6
```

Once you have integrated the `libidn` add-on into the Glibc source code directory, you are ready to configure and then build Glibc.

Configuring the Source Code

As discussed in the introduction to Chapter 7, long ago on Un*x systems far, far away, building applications that were distributed as source code usually meant manually customizing the applications' Makefiles. Things are much better now. Thanks to the magic of GNU build tools such as Autoconf and Automake, configuring, compiling, and testing Glibc on a variety of systems is both automatic and easy. You do not have to have these tools installed on your system in order to take advantage of their power—the configure script that is used to automatically configure Glibc for your hardware, operating system, and installation preferences was produced using these utilities and is included with each Glibc source code distribution.

The best way to build Glibc is to create a separate directory (referred to in the Glibc documentation as `objdir`) to hold the final and intermediate results of compiling Glibc. This makes it easy for you to compile and install Glibc without ever changing any of the files in the actual source code directory (just using them, of course). This chapter describes configuring Glibc through this mechanism, using the directory `glibc-build` as the directory where Glibc will actually be compiled.

WHERE SHOULD I PUT GLIBC?

By default, Glibc (like most applications and libraries whose configuration scripts are generated using Autoconf and Automake) will install into the appropriate subdirectories of `/usr/local` unless you specify another location for installation.

However, installing Glibc into subdirectories of `/usr/local` is the wrong thing to do unless you only want to use it for testing, and your system's load library path does not include the `/usr/local/lib` directory. The load library path contains the list (and sequence) of directories that your system's shared library loader will search for shared libraries. Your load library path may be specified in the `LD_LIBRARY` environment variable or in a configuration file such as the /etc/ld.so.conf file used on most Linux systems. If the directory `/usr/local/lib` is already in your load library path and you install a newer version of Glibc there, your system may get confused about which library to use, though `/lib` and `/usr/lib` are supposed to be searched first, regardless of whether they are present in the loader configuration file or the `LD_LIBRARY_PATH` environment variable.

In order to install Glibc in a location other than `/usr/local/lib`, you must use the configure script's `--prefix` command-line option. The syntax of this option is `--prefix=<directory>`, where `<directory>` is the directory under which the `lib` and `include` directories associated with your new version of Glibc will be created (if necessary), and where the associated components of Glibc will be installed. If you plan to install the version of Glibc that you are building as the primary C library on your system, replacing whatever version of Glibc your system is currently running, you should specify `--prefix=/usr`.

The Glibc configure script has some built-in safeguards to ensure that you actually think about where you are installing Glibc. If you actually want to install the version of Glibc that you are building in the appropriate subdirectories of the directory `/usr/local`, you will have to specify the `--disable-sanity-checks` command-line option. If you attempt to configure Glibc for installation into its default location and do not specify this command-line option to the configure script, you will see a message like the following:

```
*** On GNU/Linux systems the GNU C Library should not be installed into
*** /usr/local since this might make your system totally unusable.
*** We strongly advise to use a different prefix.  For details read the FAQ.
*** If you really mean to do this, run configure again using the extra
*** parameter '--disable-sanity-checks'.
```

Once this message displays, the configure script terminates. You must specify the `--disable-sanity-checks` option if you are not providing a specific prefix value. Note that specifying this option disables some other sanity checks for Glibc, such as whether add-ons are present, so you should use this option with care. It is a loaded gun.

To configure Glibc for compilation using a separate object directory, do the following:

1. Create the directory in which you want to build Glibc and where the intermediate object files and final libraries will be created. This directory should typically be at the same level as the `glibc-2.3.6` source directory that was extracted from the downloaded archive files, as explained in the "Downloading and Installing Source Code" section of this chapter. An example of this is the following:

   ```
   $ pwd
   ```

   ```
   /usr/local/src
   ```

```
$ ls
```

```
glibc-2.3.6
```

```
$ mkdir glibc-build
$ ls
```

```
glibc-2.3.6  glibc-build
```

2. Change your working directory to the directory you created in the previous step.

```
$ cd glibc-build
```

3. From this directory, execute the configure script found in the Glibc source directory, specifying the appropriate command-line options. The most common options that you will want to use are the `--enable-add-ons` option, to configure Glibc to be built with any add-ons that it detects in the source code directory, and the `--prefix` option. (See the sidebar "Where Should I Put Glibc?" for information about using the `--prefix` option to specify an installation location.)

An example of configuring Glibc to use any add-ons that it finds in its source directory and to eventually replace any existing version of Glibc on your system is the following:

```
$ pwd
```

```
glibc-build
```

```
$ ../glibc-2.3.6/configure --enable-add-ons --prefix=/usr
```

```
  [much output deleted]
```

■ **Note** For a complete list of the options provided by the Glibc configure script, execute the configure script with the `--help` option. For more information about all of these options, see the file named INSTALL, located in your Glibc source directory.

Once the configure script completes successfully (which can take quite a while), you are ready to proceed to the next section and actually begin building Glibc.

KERNEL OPTIMIZATION SUPPORT DURING GLIBC CONFIGURATION

Introduced early in the Glibc 2.3.x series, the configure script for Linux systems provides an optimization option, `--enable-kernel=<version>`, where *<version>* is a specific kernel version or a supported keyword. Configuring Glibc using this option strips out compatibility code that is still present in Glibc to support older versions of the Linux kernel. The majority of this compatibility code is present in a number of frequently used functions. While compiling Glibc with this option may not substantially reduce the size of Glibc, it can provide performance improvements and a reduction in code path length. To use this option, specify `--enable-kernel=current` as shorthand for the current kernel version on the configure script's command line.

Be careful! You should only configure Glibc with this option if you will never run a kernel whose version is older than the one that is running on your system when you compile Glibc. If you compiled Glibc with this option and attempt to start up with an older kernel, your system may not boot.

Compiling Glibc

Once you have successfully configured Glibc, building Glibc is easy. If you are following the suggested procedure for using a separate build directory as described in "Configuring the Source Code", you need only make sure that the build directory is your working directory and execute a command such as the following:

```
$ env LANGUAGE=C LC_ALL=C make
```

The `env` command sets the `LANGUAGE` and `LC_ALL` environment variables to appropriate values for the default locale and character set and then uses these values when executing the `make` command. The `LANGUAGE` environment variable specifies the language to use in error messages. The `LC_ALL` environment variable specifies the default font set for displaying those messages. The values shown in this sample command are appropriate for my location, which is in the continental United States. If this is not appropriate for you, you may want to change the value of these variables to reflect your default locale and character set.

■ **Tip** If you are building Glibc on a multiprocessor system with a reasonable amount of memory, you may want to take advantage of GNU make's ability to execute multiple compilations in parallel. You can do this by using the `make` command's `-j` option, followed by the number of build commands to execute in parallel.

■ **Note** If you are unfamiliar with internationalization issues, a bewildering number of potentially relevant standards and documents are available on the Web. Given that internationalization is a community by definition, a good place to start for information on open standards for internationalization under Linux is the Free Standards Group site at `http://www.li18nux.org/`. A good collection of general links on internationalization topics is available at i18nGurus.com's documentation section at `http://www.i18ngurus.com/docs`.

Once you have started building Glibc, the next step is to be patient. Depending upon the speed of your system, disk drives, and the system load, building Glibc can take quite a while, but things are improving due to hardware improvements and the amount of memory that systems have nowadays. Years ago, on an unloaded 500MHz Celeron system with 256MB of memory and running Red Hat Linux 8.0, building Glibc 2.3.2 took 4 hours and 45 minutes. Today, on an unloaded 1.7GHz Athlon system with 512MB of memory and running SUSE 10.0 Linux, building Glibc 2.3.6 is a tad faster, taking approximately 52 minutes. On a 64-bit SUSE 10.1 system with 1GB of memory, building Glibc 2.3.6 only takes 32 minutes, even with many other processes running at the same time (such as my working on this chapter).

■ **Note** If the Glibc build exits with an error when creating the ld.map file, this indicates that you are running a version of gawk that has problems in some of the `system()` calls that it uses to create and rename files. To resolve this problem, download, build, and install the latest version of gawk from `http://www.gnu.org/software/gawk/gawk.html`, and restart your build of Glibc.

Testing the Build

All Glibc source code distributions include a complete set of tests that you can use to automatically verify that your new version of Glibc has been built correctly. Testing Glibc before you install it is just plain smart, because the libraries that are built as part of Glibc, the standard C library, the loader library on Linux systems, and the Linux threads library, are the fundamental shared libraries used by all applications compiled with Glibc. This is especially critical on Linux systems, where all system applications outside the kernel depend on the standard C library and also require the loader to load shared libraries in the first place.

Once Glibc has built successfully on your system, you can automatically run all of the included Glibc tests by executing the `make check` command. The Glibc Makefile's check target compiles a number of sample applications, links them using your new version of Glibc, and then executes them to verify the correctness of the new Glibc.

Running `make check` may take a while. Once it completes successfully, you are ready to install your new version of Glibc. If it does not complete successfully, and you are sure that your system is configured correctly, you may want to consider asking a question on one of the Glibc newsgroups, or even filing a bug against Glibc as described later in the "Reporting Problems with Glibc" section of this chapter.

Installing Glibc

The instructions described in this and subsequent sections have always worked for me when upgrading Glibc—your mileage may, unfortunately, vary. Before upgrading your system, back up your critical data (never a bad idea). A Glibc upgrade does not modify your file systems other than to install new libraries, but can still render your system unbootable or unusable, because critical applications may turn out to be dependent on the vagaries of your old Glibc. Worst case, it is almost always possible to quickly revert to your previous Glibc, but the operative syllable in the word *software* is still *soft*. It is impossible to predict every scenario. Like a Boy Scout, be prepared.

Assuming that your new version of Glibc compiles correctly and passes all of its verification tests, you will be ready to install it on your system. This only takes a few minutes.

■ **Tip** Before attempting to install your new version of Glibc, especially if you have configured and compiled it to replace your system's current version of Glibc, you should first make sure that you have the tools necessary to recover from any problems encountered during the installation process. Most Linux applications use shared libraries, including basic utilities such as `ls` and ln that you will need if you experience problems during the upgrade.

The key to correcting most Glibc upgrade problems is to make sure that you have access to statically linked versions of the `ls`, `rm`, and `ln` utilities or some other statically linked program that provides the same functionality. Most Linux distributions do not provide static versions of these utilities, instead providing a utility such as BusyBox, which is (usually) a statically linked program that provides the same functionality as utilities such as `ls`, `rm`, `ln`, and a host of others. Originally developed for use on embedded Linux systems where disk space and memory is at a premium, and having a single executable that can do the job of many separate utilities is a good thing, BusyBox is also a supremely useful tool when recovering from a host of system problems.

If you cannot find a version of BusyBox on your system (in other words, if `/sbin/busybox` does not exist and the shell's `which busybox` command does not return anything), see the section titled "Using a Rescue Disk" later in this chapter. This section provides information about downloading and creating a bootable disk that you can use to resolve upgrade problems. You should, of course, do this before upgrading Glibc, unless you have another Linux system handy (or a Windows system, if you are really desperate). The remainder of the section explains how to install the version of Glibc that you have just built and tested as your primary C library, how to install it as an alternate C library for use in development or testing, and how to correct and/or recover from problems that you may experience during the installation.

Installing Glibc As the Primary C Library

This section explains how to install a new version of Glibc as a replacement for the existing version of Glibc that is installed on your system. If you have configured and built Glibc to be an alternate C library (in other words, if you configured and compiled it with no prefix or a prefix other than `/usr`), see the section "Installing an Alternate Glibc."

Assuming that Glibc has been built correctly on your system and has passed all of the tests described in the previous section, you are ready to install it on your system. Before doing so, you should take a few precautions—replacing the primary shared libraries that are used by almost all applications on your system is fairly major, as upgrades go.

Before replacing your system's existing Glibc with a new version, do the following:

- Verify that the BusyBox utility is installed on your system. If it is not, either install it or prepare a rescue disk as described in the section "Using a Rescue Disk" later in this chapter.
- Copy or otherwise back up your existing `/usr/include` directory. If the installation fails and you want to revert to your previous Glibc, you will want to be able to return to the set of include files associated with that version of the C library. You can back up your `/usr/include` directory using a command such as the following:

  ```
  $ cd /usr ; tar czvf include.tgz include
  ```

- Print a copy of your system's /etc/fstab or at least write down the disk partition corresponding to your root file system, as shown by the `df` command.

Once you have completed these precautions, upgrading your Glibc is simply a matter of executing the `make install` command and sitting back as the `make` command does the dirty upgrade work for you.

Once `make install` completes, it is quite easy to verify whether the upgrade was successful—just type **ls** or almost any other Linux command. If the command executes normally, congratulations—you have upgraded successfully.

If, instead, you receive a message such as the following, don't panic.

```
relocation error: /lib/i686/libc.so.6: symbol __libc_stack_end,
 version Glibc_2.1 not defined in file ld-linux.so.2 with link time reference
```

This sort of error means that the upgrade did not complete successfully. The first thing to check at this point is whether all of the symbolic links in `/lib/i686` and `/lib/tls` have been created correctly. This is the most common cause of problems at this point, since installing a new version of Glibc often only updates the symbolic links in `/lib`. I use the following script (available from this book's Web site) to resolve this sort of problem automatically:

```
#! /bin/sh
#
# Script to fix /lib subdir symlinks that may have been
# missed during glibc upgrade
#
# After upgrade, must be run from /lib (or /mnt/foo/lib
# if you're trying to fix things after booting from a
# rescue disk).
#
# - wvh@vonhagen.org
#
# Free for all - help yourself - it's a frigging shell-script
#

#
# Set this to whatever glibc version you're upgrading to
#
NEWGLIBC="2.3.6"

#
# fix what is/would be /lib/i686 subdir
#
if [ -d i686 ] ; then
  cp libc-$NEWGLIBC.so i686
  cp libm-$NEWGLIBC.so i686
  cp libpthread-$NEWGLIBC.so i686
  cd i686
  rm libc.so.6
  ln -s libc-$NEWGLIBC.so libc.so.6
  rm libm.so.6
  ln -s libm-$NEWGLIBC.so libm.so.6
  rm libpthread.so.0
  ln -s libpthread-$NEWGLIBC.so libpthread.so.0
  cd ..
fi

#
# fix what is/would be /lib/tls subdir
#
```

```
if [ -d tls ] ; then
  cp libc-$NEWGLIBC.so tls
  cp libm-$NEWGLIBC.so tls
  cp libpthread-$NEWGLIBC.so tls
  cp librt-$NEWGLIBC.so tls
  cd tls
  rm libc.so.6
  ln -s libc-$NEWGLIBC.so libc.so.6
  rm libm.so.6
  ln -s libm-$NEWGLIBC.so libm.so.6
  rm libpthread.so.0
  ln -s libpthread-$NEWGLIBC.so libpthread.so.0
  rm librt.so.1
  ln -s librt-$NEWGLIBC.so librt.so.1
  cd ..
fi
```

If you followed the instructions at the beginning of this section about installing BusyBox or creating a rescue disk (if necessary), it should only take a few minutes to correct the problem and have your system up (and upgraded) and running.

Installing an Alternate Glibc

The sidebar earlier in this chapter titled "Where Should I Put Glibc?" highlights the fact that you should rarely install multiple versions of Glibc on your system unless you want to test a new version before deploying it, check whether a new version of Glibc resolves execution problems that you are seeing in an application, and so on. The directory `/usr/local` is the default installation location for applications configured with automake and autoconf, which may already be in your library search path as specified in `/etc/ld.so.conf`.

When building and installing alternate versions of Glibc for testing purposes, I typically configure and install them under a directory such as `/usr/test` (which I create on my system) using the `--prefix` option (`--prefix=/usr/test` in this example) to Glibc's configure script, as discussed in the sidebar "Where Should I Put Glibc?" Using this directory highlights the fact that the versions of Glibc in this directory are only to be used for testing, and it also ensures that they are not in any directory that my system automatically searches for shared libraries.

To test applications with alternate versions of Glibc, you can execute your application with an explicit setting for the `LD_LIBRARY_PATH` environment variables that references the location where other versions of Glibc are installed. The following example illustrates using an older version of Glibc (2.3.5) installed in `/usr/test/lib` with the glibc_version program, discussed earlier in the chapter in the section titled "Identifying Which Glibc a System Is Using."

```
# ./glibc_version
```

```
2.3.6
```

```
# LD_LIBRARY_PATH=/usr/test/lib ./glibc_version
```

```
2.3.5
```

Similarly, you can write a small shell script to automate this process, as in the following example:

```
#!/bin/bash

foo=$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=/usr/test/lib:${LD_LIBRARY_PATH}

/usr/bin/ldd $*

$*

export LD_LIBRARY_PATH=$foo
```

This script first saves a copy of any setting for the `LD_LIBRARY_PATH` environment variable and then sets it to search the `/usr/test/lib` directory first. It then runs ldd (an application included with Glibc that lists shared library dependencies) on the binary to show the libraries that will be loaded, and then runs any arguments that have been passed to the script. It concludes by resetting the `LD_LIBRARY_PATH` environment variable to its original value, if any. The output from a sample run of this script looks like the following:

```
# ./glibc_wrap.sh ./glibc_version
```

```
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /usr/test/lib/libc.so.6 (0x40017000)
        /lib/ld-linux.so.2 (0x40000000)
2.3.5
```

If Glibc passes all of its internal tests during the build process, it is generally safe to directly install the new version on your system. However, if you are primarily interested in verifying that a new version of Glibc fixes problems that you are seeing in an application, installing an alternate version of Glibc for testing purposes can help you determine whether it's worth upgrading your entire system.

Using a Rescue Disk

If your system does not provide a version of BusyBox and cannot run any program that uses shared libraries after upgrading to a new version of Glibc, the easiest way to repair your system is to reboot your system from a disk known as a *rescue disk,* located on removable media such as a floppy disk or CD. Such rescue disks are freely available on the Internet and are designed to help you boot failed systems, resolve or work around common problems, and quickly restore your system to self-sufficiency. Most Linux distributions, such as SUSE, Ubuntu, Yellow Dog, and Fedora Core, support a rescue mode in their DVD edition or the first CD of their CD-based distributions. If you are using any of these distributions, you can skip this section and get a workable system by booting from the first DVD or CD in the distribution set.

If you're using another Linux distribution or simply want to have a rescue disk handy (after all, you can't have too many tools), my favorite rescue disks are the RamFloppy rescue disk (a bootable floppy-based rescue disk) and the RIP rescue disk (a bootable CD, though a floppy-based version is also available). These are available on the Internet from Kent Robotti's Web page at `http://www.tux.org/pub/people/kent-robotti/looplinux/rip/index.html`. The RamFloppy and RIP floppy rescue systems support ext2, ext3, is9660, ntfs, umsdos, ReiserFS, and VFAT file systems, making it easy for you to mount and access these types of file systems on the computer that you are having

problems with. They provide BusyBox, as well as a variety of file system management tools if you are experiencing disk corruption on your computer system. The RIP bootable rescue CD provides all of the tools on the floppy systems, but adds support for the JFS and XFS journaling file systems, as well as the fsck.jfs and xfs_repair utilities that can be used to repair corrupted file systems of those types. Another popular "rescue disk" is the latest release of the Knoppix Linux distribution (`http://www.knoppix.org`), which is a full-blown Linux distribution that boots and runs from CD. I find starting KDE and a complete GUI to be slight overkill for a rescue disk, but your mileage may vary.

To build a floppy or CD containing these rescue systems, check the README files for instructions on downloading and installing the rescue system you are interested in. To download RamFloppy and create a bootable rescue disk on a Linux system, download the file ramf-118.exe and uncompress its contents using the `unzip -L ramf-118.exe` command. You should then use `cd` to go to the `ramflop` directory, insert a blank floppy disk in your system's floppy drive, and execute the mkrescue.sh shell script to create a bootable rescue floppy. Your system must have support for the msdos file system; if the `mkrescue.sh` command complains about an unsupported file system type, try executing the `insmod msdos` command as the superuser to load the correct kernel module and then rerunning the mkrescue.sh shell script. Explaining how to recompile your kernel to add support for msdos file systems is outside the scope of this chapter.

If you are building a rescue disk on a Windows system, open a command window and execute the ramf-118.exe file to extract its contents. Change the directory to the directory created by the unzip process and execute the mkrescue.bat file to create a bootable rescue floppy for your Linux system.

Once you have created a rescue floppy or CD, you can feel quite virtuous, as though you have just purchased computer system insurance. Put the rescue disk in a safe (but memorable) place, and continue with the upgrade. See the section "Resolving Upgrade Problems Using a Rescue Disk" for information about actually using a rescue disk to repair your system, should that be necessary.

■ **Note** Most of the rescue disks available for Linux systems are for x86-based Linux systems. If you are using a PPC-based Linux distribution such as Yellow Dog, you can boot from the distribution disk and select Rescue Mode to perform the recovery procedures described later in this appendix.

Troubleshooting Glibc Installation Problems

This section focuses on resolving problems encountered after a failed upgrade to a newer version of Glibc on Linux systems, where Glibc is the primary C library.

If your system cannot run any program that uses shared libraries after upgrading Glibc, there is no need to panic—problems are generally easy to resolve once your blood pressure has returned to normal.

As explained in the note at the beginning of the section "Installing Glibc," the key to correcting most Glibc upgrade problems is to make sure that you have access to statically linked versions of the `ls` and `ln` utilities or some other statically linked program such as BusyBox that provides the same functionality. If you cannot find a version of BusyBox on your system (in other words, `/sbin/busybox` does not exist and the shell's `which busybox` command does not return anything), see the section titled "Using a Rescue Disk" for information about downloading a bootable floppy disk image and creating a boot floppy that you can use to boot your system, mount hard drive partitions corresponding to your system's / partition, and correct several symbolic links.

The next two sections explain how to correct most common Glibc upgrade problems using BusyBox or a rescue disk, respectively. The third section explains how to back out of your Glibc upgrade, if you discover that you have critical applications that no longer execute or now execute incorrectly because they are dependent on specific features of your previous version of Glibc.

Resolving Upgrade Problems Using BusyBox

If the system that you are upgrading has a copy of BusyBox installed, you are in luck—you can resolve the types of problems described in the previous section using BusyBox, usually without rebooting your computer system.

On x86 systems, installing a new version of Glibc on your system involves updating 11 primary libraries, all of which are actually symbolic links to specific versions of the associated Glibc library. On most modern Linux systems, these library links are the following:

```
/lib/libc.so.6
/lib/ld-linux.so.2
/lib/libm.so.6
/lib/libpthread.so.0
/lib/i686/libc.so.6
/lib/i686/libm.so.6
/lib/i686/libpthread.so.0
/lib/tls/libc.so.6
/lib/tls/libm.so.6
/lib/tls/libpthread.so.0
/lib/tls/librt.so.1
```

These are usually symbolic links to appropriate libraries associated with the version of Glibc that a system is running. For example, on my SUSE 10.0 system, these entries point to the following libraries:

```
/lib/libc.so.6            -> /lib/libc-2.3.5.so
/lib/ld-linux.so.2        -> /lib/ld-2.3.5.so
/lib/libm.so.6            -> /lib/libc-2.3.5.so
/lib/libpthread.so.0      -> /lib/libpthread-2.3.5.so
/lib/i686/libc.so.6       -> /lib/i686/libc-2.3.5.so
/lib/i686/libm.so.6       -> /lib/i686/libm-2.3.5.so
/lib/i686/libpthread.so.0 -> /lib/i686/libpthread-2.3.5.so
/lib/tls/libc.so.6        -> /lib/i686/libc-2.3.5.so
/lib/tls/libm.so.6        -> /lib/i686/libm-2.3.5.so
/lib/tls/libpthread.so.0  -> /lib/i686/libpthread-2.3.5.so
/lib/tls/librt.so.1       -> /lib/i686/librt-2.3.5.so
```

Similarly, on a system that has been upgraded to Glibc 2.3.6, these links would look like the following:

```
/lib/libc.so.6            -> /lib/libc-2.3.6.so
/lib/ld-linux.so.2        -> /lib/ld-2.3.6.so
/lib/libm.so.6            -> /lib/libc-2.3.6.so
/lib/libpthread.so.0      -> /lib/libpthread-2.3.6.so
/lib/i686/libc.so.6       -> /lib/i686/libc-2.3.6.so
/lib/i686/libm.so.6       -> /lib/i686/libm-2.3.6.so
/lib/i686/libpthread.so.0 -> /lib/i686/libpthread-2.3.6.so
/lib/tls/libc.so.6        -> /lib/i686/libc-2.3.6.so
/lib/tls/libm.so.6        -> /lib/i686/libm-2.3.6.so
/lib/tls/libpthread.so.0  -> /lib/i686/libpthread-2.3.6.so
/lib/tls/librt.so.1       -> /lib/i686/librt-2.3.6.so
```

Segmentation faults or errors of the following form when trying to run an application generally mean that one or more of the symbolic links for the critical Linux libraries has not been correctly updated:

```
relocation error: /lib/i686/libc.so.6: symbol __libc_stack_end,
  not defined in file ld-linux.so.2 with link time reference
```

Note that this error message indicates a discrepancy between `/lib/i686/libc.so.6` and `/lib/ld-linux.so.2`.

In most cases, to correct a failed upgrade, you simply have to set the symbolic links correctly, which can be tricky if you do not have access to utilities such as `ls` (to see what state everything is in), `rm` (to remove incorrect symbolic links), and `ln` (to create the correct symbolic links). Enter BusyBox, everybody's favorite utility in this situation!

■ **Tip** When you cannot execute any standard utility, you can still use some of the bash shell's built-in commands, most notably the `cd` built-in, to change directories, and the `echo` built-in, as a surrogate version of the `ls` command, to help you find your copy of BusyBox:

```
# cd /sbin
# echo busy*
```

```
busybox busybox.anaconda
```

Looking at the system that generated the sample error message shown previously (using BusyBox, since `ls` was broken at this point), you can examine the symbolic links:

```
# busybox ls -l /lib/libc.so.6 /lib/ld-linux.so.2 /lib/i686
```

```
lrwxrwxrwx 1 root root      14 Jan 14 17:08 /lib/libc.so.6 -> libc-2.3.6.so
lrwxrwxrwx 1 root root      12 Jan 14 17:08 /lib/ld-linux.so.2  -> ld-2.3.5.so

/lib/i686:
total 21944
drwxr-xr-x   2 root root         4096 Mar  9 09:34 .
drwxr-xr-x   9 root root         8192 Mar 22 09:45 ..
-rwxr-xr-x   1 root root   1395734 Sep  5  2002 libc-2.2.93.so
-rwxr-xr-x   1 root root 14951722 Dec  7  2002 libc-2.3.5.so
-rwxr-xr-x   1 root root 18701992 Mar  9 02:49 libc-2.3.6.so
lrwxrwxrwx 1 root root             13 Mar  9 02:50 libc.so.6 -> libc-2.3.6.so
-rwxr-xr-x   1 root root     170910 Sep  5  2002 libm-2.2.93.so
-rwxr-xr-x   1 root root   1052434 Dec  7 02:49 libm-2.3.5.so
-rwxr-xr-x   1 root root   1102227 Mar  9 02:49 libm-2.3.6.so
lrwxrwxrwx 1 root root             13 Mar  9 02:51 libm.so.6 -> libm-2.3.6.so
-rwxr-xr-x   1 root root   1037065 Mar  9 07:35 libpthread-0.10.so
lrwxrwxrwx 1 root root             18 Mar  9 09:34 libpthread.so.0 -> libpthread-0.10.so
```

In this case, the symbolic link `/lib/ld-linux.so.2` points to the wrong version of the Linux loader. You can correct this using the following commands:

```
# cd /lib
# busybox rm ld-linux.so.2
# busybox ln -s ld-2.3.6.so ld-linux.so.2
```

At this point, you should be able to execute the standard `ls` command again.

■ **Note** If one of the symbolic links created during installation is incorrect, you should check all of them to ensure that they all point to shared libraries from the same version of Glibc. In other words, if correcting the symbolic links in `/lib` corrects your problem, you should still check the symbolic links in the `/lib/i686` and `/lib/tls` directories to ensure that they point to the right files. If one of them is incorrect but others are correct, you will experience apparently spontaneous failures in some commands, while others will work fine.

Resolving Upgrade Problems Using a Rescue Disk

If you ignored my previous pleas to create a rescue disk and keep it handy during the Glibc upgrade process, I am somewhat disappointed but not surprised. I've learned how useful rescue disks can be through bitter and time-consuming experience. Rescue disks exist because things occasionally go wrong and people occasionally need to be rescued.

The section "Using a Rescue Disk" explains the idea of rescue disks and where to get my personal favorites and how to use the instructions and command scripts that come with them to produce a bootable floppy or CD rescue disk. The boot disks for most Linux distributions today provide a rescue mode that you can typically select after booting from the DVD or the first CD in the distribution set.

Since you are reading this section, you must find yourself in some circumstance where you need to use your rescue disk. To boot your system from the rescue disk, reboot your system with the rescue floppy in the floppy drive (or the rescue CD in your CD drive). If your system does not normally attempt to boot from the floppy or the CD, make sure that your system's BIOS is configured to attempt to boot from the floppy drive (usually identified as the A drive) or the CD drive before attempting to boot from your hard drive.

Once you have booted from the rescue floppy or the CD, you can log in as root (no password is necessary) and mount the partition corresponding to the root file system on your hard drive using a command such as the following:

```
# mount -t ext2 /dev/hda2 /mnt
```

You should consult the copy of your /etc/fstab file that I suggested previously that you print to identify the disk partition corresponding to your system's root partition /.

Once you've mounted the partition of your hard drive that corresponds to your system's root file system (/), you can then access the files and directories on that hard drive partition and correct the symbolic links associated with Glibc, as described in the section "Resolving Upgrade Problems Using BusyBox."

After correcting these links, change directory to */floppy-or-CD-name* and unmount the hard disk partition using a command such as the following:

```
# umount /mnt
```

You should then be able to reboot your system, which should reboot normally, using your new version of Glibc.

If you are still experiencing problems executing commands, you are probably sick of all this upgrade stuff and wish that you could just put your system back the way it was and worry about upgrading Glibc in your next lifetime. For information about backing out of an attempt at upgrading Glibc, see the next section.

Backing Out of an Upgrade

Backing out of a Glibc upgrade is very similar to correcting problems with an upgrade, except that you are changing the symbolic links associated with Glibc to point to the libraries associated with your old version of Glibc rather than ensuring that they all point to the libraries associated with your new version of Glibc. As a refresher, the basic libraries associated with Glibc are the following:

- /lib/libc-version.so
- /lib/ld-version.so
- /lib/libm-version.so
- /lib/libpthread-version.so
- /lib/librt-version.so

Builidng Glibc for your system also builds many other libraries, but these are simply installed in your `/lib` directory rather than being symlinked to from multiple locations, as are the previously named libraries. As discussed earlier in this chapter in the section "Resolving Upgrade Problems Using BusyBox," the typical symbolic links that point to these libraries are the following:

```
/lib/libc.so.6             -> /lib/libc-2.3.6.so
/lib/ld-linux.so.2         -> /lib/ld-2.3.6.so
/lib/libm.so.6             -> /lib/libc-2.3.6.so
/lib/libpthread.so.0       -> /lib/libpthread-2.3.6.so
/lib/i686/libc.so.6        -> /lib/i686/libc-2.3.6.so
/lib/i686/libm.so.6        -> /lib/i686/libm-2.3.6.so
/lib/i686/libpthread.so.0 -> /lib/i686/libpthread-2.3.6.so
/lib/tls/libc.so.6         -> /lib/i686/libc-2.3.6.so
/lib/tls/libm.so.6         -> /lib/i686/libm-2.3.6.so
/lib/tls/libpthread.so.0   -> /lib/i686/libpthread-2.3.6.so
/lib/tls/librt.so.1        -> /lib/i686/librt-2.3.6.so
```

If you are not sure what version of Glibc you were using before attempting the upgrade, you can determine which ones are available by using BusyBox or the utilities on your rescue disk to list the contents of one of the relevant directories, as in the following example:

```
# busybox ls -l /lib/i686
```

```
/lib/i686:
total 21944
drwxr-xr-x 2 root root     4096 Mar  9 09:34 .
drwxr-xr-x 9 root root     8192 Mar 22 09:45 ..
-rwxr-xr-x 1 root root  1395734 Sep  5  2002 libc-2.2.93.so
-rwxr-xr-x 1 root root 14951722 Dec  7  2002 libc-2.3.5.so
-rwxr-xr-x 1 root root 18701992 Mar  9 02:49 libc-2.3.6.so
lrwxrwxrwx 1 root root       13 Mar  9 02:50 libc.so.6 -> \
                                               libc-2.3.6.so
-rwxr-xr-x 1 root root   170910 Sep  5  2002 libm-2.2.93.so
-rwxr-xr-x 1 root root  1052434 Dec  7 02:49 libm-2.3.5.so
-rwxr-xr-x 1 root root  1102227 Mar  9 02:49 libm-2.3.6.so
lrwxrwxrwx 1 root root       13 Mar  9 02:51 libm.so.6 -> \
                                               libm-2.3.6.so
-rwxr-xr-x 1 root root  1037065 Sep  5 07:35 libpthread-0.10.so
-rwxr-xr-x 1 root root   902227 Mar  9 02:49 libpthread-2.3.6.so
lrwxrwxrwx 1 root root       18 Mar  9 09:34 libpthread.so.0 -> \
                                               libpthread-2.3.6.so
```

In this case, you can see that Glibc 2.3.6 is the current version, but that the appropriate files for Glibc 2.2.93 and Glibc 2.3.5 are also available, indicating that these versions of Glibc were previously used on your system.

■ **Note** This chapter uses BusyBox to provide most standard Linux commands. If you have booted from a CD for your Linux distribution in rescue mode or an actual rescue CD, you may not need to execute these commands by specifying `busybox`. I use these throughout to illustrate that you can do so on an otherwise application-free system, not because it is mandatory. If you can execute commands such as `ls`, `ln`, and `rm` directly, you should do so, and congratulations.

In this case, to return your system to using Glibc 2.3.5, you would use commands such as the following:

```
# cd /lib
# busybox rm ld-linux.so.2
# busybox ln -s ld-2.3.5.so ld-linux.so.2
# busybox rm libc.so.6
# busybox ln -s libc-2.3.5.so libc.so.6
# busybox rm libpthread.so.0
# busybox ln -s libpthread-0.10.so libpthread.so.0
# cd /lib/i686
# busybox rm libc.so.6
# busybox ln -s libc-2.3.5.so libc.so.6
# busybox rm libm.so.6
# busybox ln -s libm-2.3.5.so libm.so.6
# busybox rm libpthread.so.0
# busybox ln -s libpthread-0.10.so libpthread.so.0
# cd /lib/tls
# busybox rm libc.so.6
# busybox ln -s libc-2.3.5.so libc.so.6
# busybox rm libm.so.6
# busybox ln -s libm-2.3.5.so libm.so.6
# busybox rm libpthread.so.0
# busybox ln -s libpthread-0.10.so libpthread.so.0
```

At this point, you should be able to execute any of the commands that you could execute before you began your upgrade.

When reverting to a previous version of Glibc, you should also restore your previous version of the `/usr/include` directory, so that the include files therein match the version of Glibc that you are using. I asked you to back this up at the beginning of the section "Installing Glibc as the Primary C Library." To restore this, you could use commands such as the following:

```
# cd /usr
# mv include include.bad
$ tar xzvf include.tgz
```

The actual command that you have to execute depends on the name of the archive file that you created. Once the restore completes successfully, you can delete the directory associated with the failed upgrade (`include.bad` in the previous example). You may want to keep your backup archive around just in case you decide to try upgrading Glibc after whatever utility dependencies or bugs you encountered have been resolved.

Problems Using Multiple Versions of Glibc

This section highlights issues that you may encounter if you have configured, built, and installed Glibc as an alternate version of your system's C library—in other words, if you configured Glibc using its default installation prefix of `/usr/local` or any prefix other than `/usr`.

The most common case in which you may want to install an alternate version of Glibc is when you are doing development and/or testing and wish to ensure that your application works correctly with other versions of Glibc than the primary one that is installed on your system.

There are a few common problems with installing an alternate version of Glibc. These are listed in Table 12-1 in the order of the number of times that I have shot myself in the foot by using or abusing these mechanisms. Each potential problem is followed by an explanation of how to avoid it.

Table 12-1. *Common Glibc Problems and Suggested Solutions*

Problem	Solution
Accidentally compiling critical applications using a new version of Glibc and then needing to execute them before the partition where your alternate version of Glibc lives is mounted.	Do not compile applications used during the system boot process with any version of Glibc other than your primary one. If you must do this for some reason, ensure that these applications are only executed after all of your partitions are mounted, or ensure that your alternate version of Glibc is installed somewhere on your system's root (`/`) partition, which is the first partition mounted during the boot process (aside from an initial RAM disk, if you are using one).
Accidentally compiling critical applications using an alternate version of Glibc and then removing it.	Always keep backup copies of such applications, so that you can boot from a rescue disk and restore them if necessary.
Modifying the library specification files used by GCC to automatically use an alternate version of Glibc and then removing the alternate version of Glibc.	Do not do this! If you want to be able to automatically compile and link against an alternate Glibc, building a separate version of GCC that knows about the new version of Glibc is a much better idea than hacking the spec files of an existing GCC to use your new Glibc. Hacking the specification files used by an existing version of GCC is simply asking for trouble unless you are a true wizard, in which case you should not be making any other errors.

Getting More Information About Glibc

As you would expect nowadays, a wealth of additional information about Glibc is available on the Web and the Internet in general. This section provides an overview of additional sources of information about Glibc, including mailing lists where you can submit problem reports or simply ask for help if the information in this chapter does not suffice.

Glibc Documentation

Not surprisingly, the one true source of information about Glibc is its primary Web site, `http://www.gnu.org/software/libc/`. This site provides the following types of information:

- Frequently asked questions about Glibc: `http://www.gnu.org/software/libc/FAQ.html`
- General status information and links to release announcements: `http://www.gnu.org/software/libc/`
- The Glibc manual: `http://www.gnu.org/software/libc/manual`

■ **Note** To generate the manual that corresponds to the version of Glibc that you are building and installing, execute the `make dvi` command from a configured Glibc installation directory. If you are using the objdir build model (with a separate directory for your object code and configuration files), the documentation output files will still end up in the manual subdirectory of the directory where you installed the Glibc source code.

- Information about porting Glibc to other platforms: `http://www.gnu.org/software/libc/manual/html_node/Porting.html`

Other Glibc Web Sites

A simple Web search using your favorite search engine will show you thousands of messages about Glibc and Web sites that discuss every possible Glibc problem, suggestion, bug, and workaround. Besides the standard Glibc Web site (`http://www.gnu.org/software/libc/`), one Web site that always provides a good deal of information about Glibc is Red Hat's Glibc site, `http://sources.redhat.com/glibc`, which resolves to `http://sourceware.org/glibc/`.

Glibc Mailing Lists

A number of mailing lists about Glibc are hosted at Red Hat and the primary GNU Web site. You can subscribe to these lists at `http://sources.redhat.com/glibc/` or `http://www.gnu.org/software/libc/`.

Mailing lists related to Glibc are the following:

- *bug-glibc:* A relatively high-traffic list to which you should report problems with Glibc. You can view archives of this list at `http://sources.redhat.com/ml/bug-glibc/`.
- *glibc-cvs:* A relatively high-traffic list showing modifications to the Glibc source code archive stored in the CVS source code control system. You can view archives of postings to this list at `http://sources.redhat.com/ml/glibc-cvs/`.
- *libc-alpha:* A list that discusses issues in porting and supporting Glibc on a variety of platforms. You can view archives of posts to this list at `http://sources.redhat.com/ml/libc-alpha/`.
- *libc-announce:* A low-traffic list to which announcements of new releases of Glibc are posted. You can view archives of posts to this mailing list at `http://sources.redhat.com/ml/libc-announce/`.
- *libc-hacker:* A closed list in which details of Glibc development and porting are discussed by the Glibc maintainers. Mere mortals cannot post to this list, but can view archives of postings to this list at `http://sources.redhat.com/ml/libc-hacker`.

Reporting Problems with Glibc

As mentioned in the previous section, you should report Glibc problems to the bug-glibc mailing list. You can submit problem reports via e-mail to `bug-glibc@gnu.org`. You can also view archives of postings made to this list at `http://sources.redhat.com/glibc`. This is the best and easiest-to-use source of information about known problems in different versions of Glibc. You should check this database before submitting a problem report, not only to avoid duplication, but also to see if the problem that you are experiencing has already been fixed.

Moving to Glibc 2.4

Version 2.4 is the first major release in a very long period of time, and therefore introduces changes to the C API as well as a new application binary interface (ABI). Because the 2.4 version of Glibc is new, and due to its ABI changes, the stable 2.3.x release series continues to be supported and provides the current standard, widely deployed ABI. However, 2.4 is the up-and-coming version, so you may want to upgrade (or test an upgrade) to see how it works, as well as upgrade for the reasons mentioned at the beginning of this chapter.

For Linux systems, version 2.4 of the GNU C library is intended for use with Linux kernel version 2.6.0 and later. The only threading implementation supported in Glibc 2.4 is the Native POSIX Threading Library (NPTL). Most of the C library will continue to work on older Linux kernels, and many programs will still run correctly with older kernels as along as they do not use Pthreads or related threading calls. NPTL was designed to be compatible with LinuxThreads, but this cannot be guaranteed. The LinuxThreads add-on implementation of Pthreads for older Linux kernels is no longer supported and is no longer updated or distributed with Glibc releases as of 2.4.

When building and testing Glibc 2.4, Linux kernel versions prior to 2.6.16 have some bugs that may cause some of the Pthread-related tests executed by the `make check` command to fail. If you see such problems, consider upgrading your kernel before proceeding to deploy Glibc 2.4.

Version 2.4 of the GNU C library supports these configurations for using Linux kernels:

- alpha*-*-linux-gnu (requires Linux 2.6.9 for NPTL)
- i[34567]86-*-linux-gnu
- powerpc-*-linux-gnu
- powerpc64-*-linux-gnu
- s390-*-linux-gnu
- s390x-*-linux-gnu
- sh[34]-*-linux-gnu (requires Linux 2.6.11)
- ia64-*-linux-gnu
- sparc*-*-linux-gnu
- sparc64*-*-linux-gnu
- x86_64-*-linux-gnu

The code for other CPU configurations supported by volunteers outside of the core Glibc maintenance effort is contained in the separate ports add-on. You can find glibc-ports-2.4 distributed separately on the same Web or FTP site from which you downloaded the main Glibc distribution files. The following configurations are known to work using the ports add-on:

- arm-*-linux-gnu (requires Linux 2.6.15 for NPTL; no SMP support)
- arm-*-linux-gnueabi (requires Linux 2.6.16-rc1 for NPTL; no SMP)
- mips-*-linux-gnu (requires Linux 2.6.12 for NPTL)
- mips64-*-linux-gnu (requires Linux 2.6.12 for NPTL)

The ports distribution also contains code for other configurations that do not work or have not been maintained recently but will be of use to anyone trying to make a new configuration work. If you want to take a crack at a port, contact the Glibc maintainers.

CHAPTER 13

Using Alternate C Libraries

Chapter 12 discussed building and installing the latest version of the GNU C library, popularly known as Glibc, or simply *glibc* to its friends. Applications built with the vanilla GCC compilers use Glibc by default, which is usually a good thing from the completeness and simplicity points of view, but Glibc isn't the only C library game in town. This chapter explores the more popular alternate C libraries that are available for Linux, explaining where to find them, how to build them, and most importantly, why you might want to bother building an alternate C library when you've already got a perfectly good GNU C library on your development system.

Why Use a Different C Library?

Glibc is the unseen force that makes GCC, most C language applications compiled with GCC on Linux systems, and all desktop GNU/Linux systems themselves work. Note that I said *most C language applications* and *desktop* Linux distributions. If you're developing for a desktop Linux distribution, chances are that some other application that's running already uses Glibc, so why not leverage the fact that it's probably already loaded into memory and use it with your application(s)? After all, that's the beauty of the shared library concept, since all applications can share the same in-memory version of Glibc. However, in cases such as consumer electronics devices and embedded computing in general, where you have a limited amount of memory and other resources, you want to do whatever you can to reduce the size of your applications and the amount of memory that they require to execute.

The main problem with Glibc is that it is really, really big. Because Glibc provides the infrastructure for all C applications, Glibc has to provide a huge collection of function and symbol definitions even if you are actually only using some of them. The fact that Glibc is usually used as a shared library saves you some initial application size, but as soon as you actually load the library into memory, you are getting the same worst-case effect (application size + library size) as if you had statically linked your application in the first place and used everything in Glibc. As I mentioned in the previous paragraph, the shared library concept at least guarantees that all applications that use Glibc share the same in-memory copy—but what if you don't need everything that is in Glibc?

The classic solution for reducing the size of an application that uses shared libraries is to unroll the library and recreate it so that it only contains the functions that you actually use. Doing this eliminates anything that's unnecessary and which therefore needlessly increases the memory footprint of your application, while still leveraging the shared library concept. Unfortunately, in real life this is both hard to do and extremely time-consuming, though some open source tools can help with this. Unfortunately, almost every application that is more complex than hello, world uses different standard C functions, and you would therefore have to repeat the "link, identify problems, repeat" cycle a good many times for each of your applications. That is a possibility when there is a small number of applications, but it is a maintenance nightmare waiting to happen when you update or upgrade your application and you need to remember the application's original requirements and repeat the process to account for any new requirements that you have introduced in your updated application code.

It is tempting to think that the size of Glibc does not affect your application if you are writing in a language other than C, but sorry—Glibc is also linked with applications compiled with any of the other GCC compilers. Ada, C++, Fortran, Java, and Objective-C all leverage Glibc due to the complete set of low-level functionality that it provides.

I am sure that you see where I am going with all of this. The best and easiest alternative to linking applications with Glibc that helps reduce application size (and therefore, required runtime resources) is to statically link applications with an alternate C library. These C libraries are smaller than Glibc. This is achieved through increased configurability, implementation of a subset of the functions provided by Glibc, or both. The next section introduces the most common Glibc alternatives used today, discusses how they achieve a smaller footprint, and explains the way(s) in which each is often used.

Overview of Alternate C Libraries

There are four primary alternatives to Glibc. Alphabetically, these are the following:

- *dietlibc*: A small C library optimized for size, written by Felix von Leitner with contributions by many others. dietlibc is easy to build and use in both native and cross-compiled settings because it provides its own driver program (called diet) that runs gcc and maps in dietlibc as the C library. The dietlibc home page is located at `http://www.fefe.de/dietlibc/`.
- *klibc*: A small C library subset that is designed for use during the Linux boot process (and specifically with the initramfs filesystem introduced to mainline Linux distributions with the 2.6 Linux kernel). klibc was written by H. Peter Anvin of ISOLINUX/SYSLINUX fame. klibc did not really have a home page at the time that this book was written, but you can always retrieve the latest version from `ftp://ftp.kernel.org/pub/linux/libs/klibc/`.
- *Newlib*: A small C library, which includes an equally small math library, designed for use on embedded systems. Newlib was originally written by the folks at Cygnus Software, which was acquired by Red Hat long ago. Newlib is now maintained by Jeff Johnston and Tom Fitzsimmons at Red Hat. The Newlib home page is located at `http://sources.redhat.com/newlib/`.
- *uClibc*: The best known and most complete alternate C library designed for use on embedded systems and maintained by Erik Anderson of BusyBox fame (or perhaps of uClibc fame, depending on your perspective). The uClibc home page is located at `http://www.uclibc.org`. UClibc has a very friendly configuration interface and is easy to build for cross-compiled toolchains thanks to its integration with the buildroot project (`http://www.buildroot.org`).

The remainder of this chapter discusses how to work with and use each of these. Of course, the hard part of building these is building them so that they work with cross-compilers, but don't worry—I'll cover that in Chapter 14.

■ **Note** Other open source C libraries are also available, most notably those used by BSD-inspired distributions such as FreeBSD, NetBSD, OpenBSD, and Apple's Darwin. However, I see many more people using the four C libraries I listed previously—and I am more of a Linux guy—so I have chosen to focus on those.

Overview of Using Alternate C Libraries

Before plunging into discussing my favorite alternate C libraries, it is worth reviewing how to tell GCC compilers to link with C libraries other than the standard C library. The key to this is the `-nostdlib` option, which is supported by all GCC compilers. This option tells your GCC compiler not to use the standard system startup files or libraries when linking. When you specify this option, only the object

files and libraries that you specify on the command line will be passed to the linker. This means that you generally have to supply your own startup routine, such as dietlibc's `/opt/diet/lib`*foo*`/start.o` or klibc's *installdir*`/klibc/arch/platform/crt0.o`, and that you must explicitly supply the name of the alternate C library that you are linking with on your GCC command line.

■ **Note** One of the standard libraries bypassed by the `-nostdlib` option is libgcc.a, a library of internal routines that provides special functions for various languages and workarounds for problems on specific types of machines. You will therefore usually need to specify the `-lgcc` option on the command line (or in the `LDFLAGS` environment variable) for any application that you compile using `-nostdlib`.

You must also remember to add the alternate C library's include directories to the beginning of your include path, so that its include files are found before any generic system versions are encountered. Finally, you must also remember to define any special symbols required by the alternate C library. For example, dietlibc requires that you define `__dietlibc__`, while klibc requires that you define `__KLIBC__`.

As an example, the following command line tells gcc to compile a sample hello, world and link it using dietlibc rather than Glibc on a sample 64-bit system:

```
$ gcc -nostdlib -static -L/opt/diet/lib-x86_64 ➥
    /opt/diet/lib-x86_64/start.o -o foo hello.c ➥
    -isystem /opt/diet/include -D__dietlibc__ ➥
    /opt/diet/lib-x86_64/libc.a -lgcc
```

Most alternate C libraries either provide wrappers for gcc (dietlibc) or can be directly integrated into gcc during the build process (Newlib, uClibc). You can always convince gcc to use an alternate C library during compilation with the right combination of options and include directories. Subsequent sections of this chapter explain how to do this for each of the alternates to Glibc that are designed to be built as stand-alone libraries.

Building and Using dietlibc

dietlibc is one of my favorite small C libraries and includes a threading implementation and a math library. It is extremely small and easy to build and cross-compile, and has been tested and used on many platforms. dietlibc is designed for use as a static library, not a shared library, though it can be built as a shared library and used that way if you insist. But given the incredibly small size of statically linked binaries that use dietlibc, I would say "why bother?"

Building dietlibc also builds an extremely convenient driver program for gcc in both native and cross-compiled modes that causes gcc to link with dietlibc rather than Glibc, so there is no need to rebuild your existing toolchains, whether they provide cross or native compilers. dietlibc has been tested and used on the Alpha, ARM, HP/PA, IA-64, i386, MIPS, s390, SPARC, SPARC64, PPC, and x86_64 platforms.

■ **Caution** If you are thinking of using dietlibc in your projects, one thing that you need to be aware of is that dietlibc is GPL (i.e., released under the GNU General Public License), not LGPL (i.e., released under the GNU Lesser Public License). I am not a lawyer (thank goodness!), but in a nutshell, this means that code that links with dietlibc is also GPL. Glibc is LGPL so that it can be used by proprietary code without that code becoming GPL. This is not the case with dietlibc. For more information about licensing and dietlibc, contact the dietlibc folks on their site, `http://www.fefe.de/dietlibc/`.

For more information about dietlibc, there is an enlightening and entertaining presentation by its author at `http://www.fefe.de/dietlibc/talk.pdf`. It is a talk that was originally presented in 2001 at the Linux-Kongress in Germany. It is well worth a read if not just for its amusing observations regarding Glibc, BSD, and Linux licensing and coding philosophy.

Getting dietlibc

The easiest way to get the latest dietlibc is to retrieve it from its online source code repository, which is stored in CVS, a popular source code control system. To retrieve dietlibc, create a working directory to hold its source code and your notes, and then issue the following command:

```
$ cvs -d :pserver:cvs@cvs.fefe.de:/cvs -z9 co dietlibc
```

This creates the directory `dietlibc` on your system and populates it from the source code archive on the dietlibc CVS server. Depending on the speed of your Internet connection, retrieving the source code may take a few minutes.

Building dietlibc

After retrieving the dietlibc source code as explained in the previous section, change the directory into the `dietlibc` directory created by retrieving the source code, and type **make**. The dietlibc build process automatically detects your CPU type and creates the appropriate subdirectories for your CPU and architecture. Because dietlibc has very few external dependencies, it usually compiles cleanly, regardless of the type of processor that you are using.

To cross-compile dietlibc, you specify the same type of environment variables that you do when cross-compiling the Linux kernel: `ARCH` and `CROSS`. For example, to cross-compile dietlibc for the MIPS platform using a GCC cross-compiler with the prefix `mipsel-linux-`, you would execute the following command:

```
$ make ARCH=mips CROSS=mips-linux- all
```

The dietlibc Makefile provides built-in shortcuts for the ARM, Alpha, MIPS, PPC, SPARC, and i386 platforms, enabling you to cross-compile by simply typing a command such as `make mips`. The dietlibc Makefile assumes that your cross-compilers are in your `PATH`, and that the components of your GCC cross-compiler follow the standard naming convention used by GCC-based cross-compilers, which is *architecture-operatingsystem-application*. For example, the version of gcc that produces output that runs on MIPS Linux systems is typically named mips-linux-gcc.

■ **Tip** Though dietlibc comes with a number of shortcuts, they don't always agree with the standard architecture naming conventions used by cross-compilation build tools such as crosstool (discussed in Chapter 14). To synchronize the two, you can create symbolic links in the dietlibc source directory with the "right" architecture names. As a specific example, a standard crosstool target is powerpc, which produces a gcc cross-compiler named powerpc-linux-gcc, while dietlibc seems to expect ppc-linux-gcc in its Makefile. To resolve this, I simply created a symbolic link named *powerpc* to the `ppc` directory in the dietlibc source directory. Voila! No problems.

Once compilation has successfully completed, you can install dietlibc on your system by creating the directory `/opt/diet` and then executing the `make install` command. The C libraries for each target platform will be installed in directories containing the name of the target, such as `/opt/diet/lib-powerpc`, `/opt/diet/lib-x86_64`, and so on. Programs that run on the host system (i.e., the system on

which you compiled dietlibc, regardless of whether you were cross-compiling or not) will be installed in /opt/diet/bin.

Using dietlibc with gcc

As mentioned previously, building dietlibc also builds a front end for gcc that causes gcc to build against the header files used by dietlibc and to link with dietlibc rather than Glibc (or some other C library). This front-end application is known as diet. To use it, simply precede your standard gcc command line with the name of the diet application, as in the following example, which shows building a static version of the standard hello, world application with both Glibc and dietlibc, and the resulting difference in binary size:

```
$ gcc -o hello hello.c -static
$ diet gcc -o hello_diet hello.c -static
$ ls -l hello hello_diet
```

```
-rwxr-xr-x  1 wvh users 521447 2006-04-06 13:21 hello
-rwxr-xr-x  1 wvh users   2783 2006-04-06 13:20 hello_diet
```

Though this is obviously not a complex program, the difference in size is pretty remarkable. When statically linked with dietlibc, the same binary, you save over half a megabyte. We have a winner!

Using dietlibc with a cross-compiler is similarly easy, as shown in the following example. The following example shows using a PowerPC cross-compiler to build a static version of the standard hello, world application with both uClibc and dietlibc, demonstrates that both are ppc executables, and then shows the resulting difference in binary size:

```
$ powerpc-linux-gcc -o hello_ppc hello.c --static
$ diet powerpc-linux-gcc -o hello_ppc_diet hello.c -static
$ file hello_ppc hello_ppc_diet
```

```
hello_ppc:      ELF 32-bit MSB executable, PowerPC or cisco 4500,➥
    version 1 (SYSV), statically linked, not stripped
hello_ppc_diet: ELF 32-bit MSB executable, PowerPC or cisco 4500,➥
    version 1 (SYSV), statically linked, not stripped
```

```
$ ls -al hello_ppc hello_ppc_diet
```

```
-rwxr-xr-x  1 wvh users 22338 2006-04-06 13:51 hello_ppc
-rwxr-xr-x  1 wvh users  2492 2006-04-06 14:08 hello_ppc_diet
```

You can also use dietlibc with programs that use an Autoconf/Automake-driven approach to configuration, as in the following example:

```
$ CC="diet powerpc-linux-gcc" ./configure --disable-nls
```

If you experience problems using dietlibc, you can add its -v switch to produce verbose output that may help you identify the source of any problems. dietlibc has been used for quite a while, on a number of platforms, and is quite stable. If you actually find a problem, I am sure that Felix would love to hear about it.

Building and Using klibc

klibc is a small C library subset that was designed as a small C library that could be used by programs running from an initramfs-style initial RAM disk during the Linux boot process. Though not designed as a general-purpose C library replacement, it is discussed here because many embedded systems do not provide persistent storage, and therefore run all of their applications from a RAM disk of one sort or the other. Traditionally, the Linux kernel loaded an initial RAM disk from a compressed ext2 filesystem image known as an *initrd* which was bundled with the kernel; but newer 2.6 kernels have changed to a cpio-formatted initramfs image, which the kernel can load into and access directly from its page and directory caches. Look, Ma, no block device! (For more information about initramfs, see `http://www.linuxdevices.com/articles/AT4017834659.html`.) Given that the new model for an initial set of utilities used during the boot process is initramfs, which uses klibc, it is worth discussing klibc here because you may want to use it in your embedded projects. The initial announcement of the current klibc project is `http://lwn.net/Articles/7117/`.

As you might gather from the previous paragraph, klibc has been designed on Linux systems and has only been extensively tested there. It has been used and tested on Linux systems running on a wide variety of architectures, including Alpha, ARM, i386, ia64, MIPS, PA-RISC, PPC, PPC64, s390, SPARC, and x86_64. You are certainly welcome to try to build and use it on systems other than Linux, but you may not be able to get a lot of help with any problems that you encounter.

■ **Caution** If you are thinking of using klibc in your projects, one thing that you need to be aware of is that parts of klibc are GPL. This means that code that links with klibc is also GPL. You should think twice before using klibc as the only C library in a proprietary, commercial product, or at least warn your company that you are probably giving away all of the intellectual property in your project.

Here are some useful klibc resources on the Web:

- `http://www.zytor.com/mailman/listinfo/klibc/`: A mailing list for klibc and early-userspace issues.
- `http://www.zytor.com/cvsweb.cgi/klibc/`: A CVSWeb repository where you can view the latest changes to klibc.
- `rsync://rsync.kernel.org/pub/linux/libs/klibc/cvsroot/`: A location where you can rsync the latest checked-in changes from the repository. (Note that this is an rsync URL, not a browser URL.) The CVS archive for klibc was not directly accessible via the Web at the time that this book was written.

Getting klibc

You can always retrieve the latest version of klibc from `ftp://ftp.kernel.org/pub/linux/libs/klibc/`. The latest version at the time that this book was written was klibc 1.3.

Building klibc

Building klibc is relatively simple. After retrieving the archive for the latest version (klibc-1.3.tar.bz2, in this example), unpack it using a command such as the following:

```
$ tar jxvf klibc-1.3.tar.bz2
```

This creates a directory named `klibc-1.3`. Change directory into that directory, and create a symbolic link named `linux` in this directory that points to the location of a directory containing the configured kernel source for your target version of Linux.

CONFIGURING THE LINUX KERNEL SOURCE FOR A 2.6 KERNEL

If you are running a custom-built kernel, you have already configured it and the directory that contains your kernel source is correctly configured. However, if you are using a Linux system that came with a prebuilt kernel, you will probably need to configure your kernel source manually in order to provide the configuration information and symbolic links required for building klibc.

Most 2.6-based Linux distributions provide a copy of the kernel configuration file used to configure the kernel in their `/boot` directory. This file generally has a name of the form config-*version*, where *version* is the value returned by the command `uname -r`. Once you have located the correct file, copy it to the directory containing the source code for your kernel, giving the copy of the configuration file the name .config. This is usually identified by a symlink named `/usr/src/linux`, which typically points to a specific version of the Linux kernel source located in the directory `/usr/src`. As an example

```
# pushd /usr/src
# ls -lg
```

```
total 9
lrwxrwxrwx  1 root   17 2006-02-19 18:38 linux -> linux-2.6.13-15
drwxr-xr-x  3 root   72 2005-08-27 16:20 linux-2.6.11.4-20a
drwxr-xr-x 18 root 1120 2005-09-11 20:47 linux-2.6.11.4-21.8
drwxr-xr-x 18 root 1056 2005-11-10 14:28 linux-2.6.11.4-21.9
drwxr-xr-x  3 root   72 2006-02-19 18:38 linux-2.6.12-5.7
drwxr-xr-x 19 root  840 2006-04-06 14:53 linux-2.6.12-5.8
drwxr-xr-x 18 root 1024 2006-01-04 05:23 linux-2.6.13-15
```

```
# uname -r
```

```
2.6.13-15-default
```

```
# cp /boot/config-2.6.13-15-default linux/.config
# cd linux
# make oldconfig ; make prepare
```

The last two commands will produce a great deal of output, but your kernel source directory will be correctly configured for building klibc when the commands complete. Note that this process does not actually build the kernel but merely creates the correct configuration files and symbolic links in the kernel source directory so that you can subsequently build the kernel.

Next, simply type **make**, and klibc will build, displaying output such as the following:

```
$ make
```

```
  GEN     klcc/klibc.config
  GEN     klcc/klcc
  HOSTCC  scripts/basic/fixdep
  GEN     klibc/syscalls/SYSCALLS.i
  GEN     klibc/syscalls/syscalls.nrs
  GEN     klibc/syscalls/syscalls.mk
  KLIBCAS klibc/syscalls/_exit.o
  KLIBCAS klibc/syscalls/__clone.o
  KLIBCAS klibc/syscalls/fork.o
  KLIBCAS klibc/syscalls/vfork.o
  KLIBCAS klibc/syscalls/getpid.o
[much output deleted]
  KLIBCLD utils/shared/uname
  KLIBCCC gzip/gzip.o
  KLIBCCC gzip/util.o
  KLIBCCC gzip/unzip.o
  KLIBCCC gzip/inflate.o
  KLIBCLD gzip/gzip
  LN      gzip/gunzip
  LN      gzip/zcat
```

Note the similarities to kernel build system output, which makes sense because klibc uses the same kbuild subsystem. Because klibc shares the same build system with the kernel, you can easily cross-compile klibc by specifying the standard `ARCH` and `CROSS` environment variables on your make command line, as in the following example:

```
$ ARCH=ppc CROSS=powerpc-linux- make
```

You'll also notice that building klibc compiles a variety of utilities. It actually compiles these statically using shared libraries for your convenience when creating a cpio archive for use as an initramfs filesystem.

Finally, you can test klibc by executing the command `make test`, which will build a number of test programs located in the `klibc/test` subdirectory of your build directory.

Using klibc with gcc

As an explicit example of compiling an application statically with klibc, the following is a script I used to compile and link a sample hello, world application on a 64-bit test system where I had built the latest klibc in the directory `/home/wvh/klibc/klibc-1.3`. Obviously, you would ordinarily encapsulate this sort of process inside a Makefile, but it's useful to see all of the object files, libraries, and include paths required once I eliminated the standard system startup file, C library, and include directories by specifying `-nostdlib` and `-nostdinc`.

```
$ gcc -nostdlib -nostdinc -static -D__KLIBC__ \
    -I/home/wvh/klibc/klibc-1.3/include/arch/x86_64 \
    -I/home/wvh/klibc/klibc-1.3/include/bits64 \
    -I/home/wvh/klibc/klibc-1.3/include \
    -I/usr/lib/gcc/x86_64-unknown-linux-gnu/4.2.0/include \
    -I/home/wvh/klibc/klibc-1.3/linux/include \
    -c hello.c
```

```
$ ld hello.o \
    /home/wvh/klibc/klibc-1.3/klibc/arch/x86_64/crt0.o \
    /home/wvh/klibc/klibc-1.3/klibc/libc.a \
    -o hello_klibc
```

When statically linked using Glibc, my hello, world program on this platform was 521447 bytes in size. When statically linked with klibc, the same binary was 15345 bytes.

Your pathnames and architecture directory will almost certainly differ from those in this example, but will still be analogous. Also, the include directories that you need to search will probably also differ because these depend on the functions that you are using in your application, and all I used was `printf()`. Building with klibc will certainly be a useful part of your gcc toolbox in the future if you are building utilities that you want to include in an initramfs cpio archive.

Building and Using Newlib

Newlib is a collection of software from a variety of sources and provides a small C library and accompanying math library that were designed for use on embedded systems. Newlib was originally put together by the folks at Cygnus Software, an early pioneer of using Linux on embedded systems that was acquired by Red Hat long ago.

■ **Note** Newlib is released under a combination of licenses. Its licensing terms are typically described as being "BSD style." (The actual license can be found at the URL `http://sourceware.org/newlib/COPYING.NEWLIB`.) This means that you do not need to distribute the source code for anything that you link with Newlib, and that you can therefore use Newlib in your projects without your code becoming GPL. Please also note that I am not a lawyer, so if you want to use Newlib in a commercial product, you should still make sure that your legal department reviews its licensing terms and is happy with them (i.e., able to bill you extensively for this service).

Over the years, many people have used Newlib to develop a variety of embedded toolchains, and it has been quite popular for a while. To be honest, I am not a fan. Newlib was designed and developed for use with cross-compilers and therefore has very narrow definitions of acceptable host and target platform names, which I find infuriating to deal with, especially since it is not easy to extract a list of supported platform names. The most obvious case of this for me was simply trying to build Newlib on a desktop Linux system so that I could link with its C library to produce binaries that I can deploy on a target x86-based system. In order to do this easily from the 64-bit system that I happened to be using, I basically had to build the whole toolchain, starting with the GNU binutils and GCC itself, so that they had the right prefixes and names so that I could then build Newlib with the right target specification.

On the other hand, Newlib is extremely easy to integrate with GCC, which provides the `-with-newlib` option for that very purpose. Once I finished building everything manually (which just took some time), the Newlib-enabled version of gcc (the compiler I was using) worked smoothly, and the tight integration between Newlib and gcc meant that I did not have to do any special magic to update library and include paths or manually integrate a custom startup routine when building my applications. Your mileage may vary.

Getting Newlib

The easiest way to get the latest Newlib source code is to retrieve it from its online source code repository, which is stored in CVS. To retrieve Newlib, create a working directory to hold its source code and your notes, and then issue the following command to log into the remote CVS server:

```
$ cvs -z 9 -d :pserver:anoncvs@sources.redhat.com:/cvs/src login
```

```
Password:
```

Enter **anoncvs** when prompted for the password and press Enter. You can then enter the following command to check out the Newlib source code:

```
cvs -z 9 -d :pserver:anoncvs@sources.redhat.com:/cvs/src co newlib
```

This creates the directory `src` on your system and populates it from the source code archive on the Newlib CVS server. Depending on the speed of your Internet connection, retrieving the source code may take a few minutes.

The previous example retrieves the latest version of the Newlib source code. You can retrieve archives of the source code for specific releases of Newlib from the Newlib FTP directory, which is located at `ftp://sources.redhat.com/pub/newlib`. The latest official Newlib release at the time that this book was written was Newlib 1.14.0, which was released in December 2005.

Building and Using Newlib

As I discussed previously, it is challenging to build Newlib stand-alone for a host system. Since this isn't the way in which one would ordinarily use Newlib, I won't go into the process here. Building and using Newlib is usually done within the context of a cross-compiler build. Building cross-compilers is discussed in Chapter 14.

Building and Using uClibc

The uClibc (pronounced you-see-lib-see) C library is the best known, most complete, and most extensively used alternate C library. The uClibc project's home page is located at `http://www.uclibc.org`, and is brought to you by Erik Anderson, the patron saint of embedded programmers. Originally developed for use on embedded systems, where the bloat factor of Glibc made it unattractive (or impossible) to use, uClibc is also extensively used on live CDs or rescue disks, where binary size can be equally significant. The *u* in uClibc is technically the Greek character μ , which is generally used to mean "micro," but nobody (including myself) wants to continually insert gonzo characters in otherwise normal text, so I'll just call it uClibc like everyone else does. My apologies to the purists.

■ **Note** Like Glibc itself, uClibc is released under the LGPL. Glibc and uClibc are LGPL so that they can be used by proprietary code without that code becoming GPL. Please also note that I am not a lawyer and would find that suggestion to be insulting.

The uClibc C library runs on both standard and MMU-less processors (i.e., processors without a memory management unit) and has been extensively tested on processors such as the Alpha, ARM, cris, i386, i960, h8300, m68k, MIPS/MIPSEL, PowerPC, SH, SPARC, and v850.

UCLIBC AND UCLINUX—WHAT'S ALL THIS, THEN?

Linux for MMU-less processors was traditionally known as µClinux, (however, I will use the common spelling with a lowercase *u* throughout) and many people confuse this with uClibc because of the fact that uClibc is frequently used in conjunction with uClinux. The uClinux project (Linux/Microcontroller) was traditionally maintained as a huge set of patches outside of the mainline Linux kernel source; but one of the more significant aspects of the 2.6 Linux kernel was the merger of the then current uClinux patches into the mainline kernel source, which simplified life for everyone. Unfortunately, uClinux patches to the mainline kernel continue to proliferate as external patch sets (in their defense, it is not always easy to get processor-specific patch sets integrated into the mainline kernel), so the same confusion continues to exist. See the uClinux home page (`http://www.uclinux.org`) and the uClinux developer's forum (`http://www.ucdot.org`) for more information on the latest uClinux patches.

My apologies to Martin Luther King Jr., but I have a dream that one day on the Internet the uClinux patches and the mainline Linux kernel will be able to sit down together at the table of brotherhood. Until then, if you are working with an MMU-less processor, you should always check the uClinux sites (and your silicon vendor's site) for any processor-specific Linux patches for your hardware. You may also want to check out TimeSys' LinuxLink subscriptions (`http://linuxlink.timesys.com`), whose goal is to provide embedded Linux developers with the latest, most up-to-date kernel patches, uClinux or not, and toolchains for their hardware.

One of the coolest aspects of uClibc is its configuration mechanism, which supports both command-line and terminal-oriented configuration. This configuration mechanism shares configuration file syntax with the 2.6 Linux kernel, so its configuration files are easy and familiar to work with. I'll use the terminal-oriented configuration mechanism in the examples later in this section; but people who insist on working on a DecWriter or an LA120 can still configure the kernel in command-line mode (though you are going to waste a box or two of paper).

Getting uClibc

You can get the latest uClibc source in a variety of ways: either via FTP, using the subversion SCCS (Source Code Control System), or as part of the more ambitious buildroot project discussed in detail in Chapter 14. buildroot provides an easy way to build uClibc and a cross-compiler that uses it (automatically retrieving the latest source code for all components), build a root filesystem using that cross-compiler, schedule veterinary appointments, and send flowers to your mother on significant holidays. This chapter focuses on alternate C libraries, so I'll just discuss retrieving, building, and using uClibc here.

You can retrieve the latest official release of uClibc via FTP at `http://www.uclibc.org/downloads`. The latest official release at the time that this book was written was uClibc 0.9.28, which dates from mid-2005, so official releases are not exactly flying out of uClibc.org. The archive file containing uClibc 0.9.28 is named uClibc-0.9.28.tar.bz2. After retrieving this file, you can extract its contents by using the following command (which will create the directory `uClibc-0.9.28`):

```
$ tar jxvf uClibc-0.9.28.tar.bz2
```

A much better solution for getting the latest and greatest uClibc is to use subversion to retrieve the latest source, using the following command:

```
svn co svn://uclibc.org/trunk/uClibc
```

This automatically checks out a copy of the latest source code for you, creating a directory named `uClibc` on your system. You can subsequently update your copy of the source by issuing the `svn update` command from the same directory where you checked out the uClibc source tree.

Building uClibc

This section explains how to configure and build uClibc for a sample processor (i386). The uClibc configuration mechanism makes it easy to configure every nuance of uClibc for any architecture and processor family. However, since these are so architecture- and processor-specific, I will just highlight the most critical configuration screens in this section. I strongly encourage you to explore the screens that aren't discussed in detail in this section—uClibc is highly configurable, and selecting the right set of processor-specific configuration parameters can both reduce the size of the resulting uClibc library and increase its performance.

■ **Caution** The recommended way to build and use uClibc is by building a uClibc toolchain, which is itself most easily done by building buildroot, which is discussed in Chapter 14. This section explains how to configure and build uClibc stand-alone, which is not the recommended way of using uClibc. However, even when using buildroot, it is possible and useful to configure uClibc directly, so the information in this section is useful, regardless.

The easiest way to build uClibc is to execute the `make menuconfig` command from the directory that you check out from subversion or that was created by extracting the contents of an archive file that you retrieved via FTP. After executing the `make menuconfig` command, a screen like the one shown in Figure 13-1 displays.

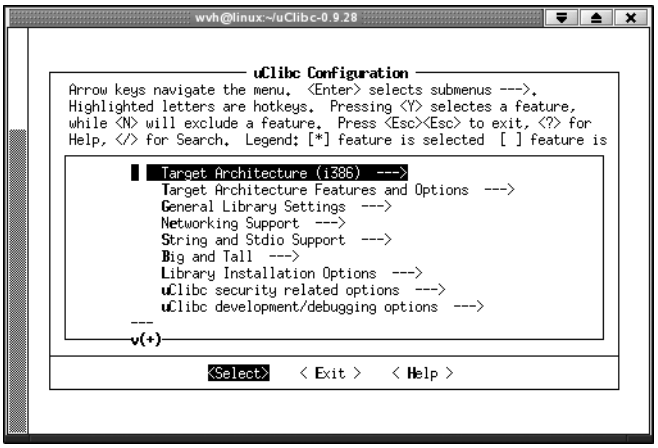

wvh@linux:~/uClibc-0.9.28
uClibc Configuration
Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> selectes a feature,
while <N> will exclude a feature. Press <Esc><Esc> to exit, <?> for
Help, </> for Search. Legend: [*] feature is selected [ ] feature is
Target Architecture (i386) --->
Target Architecture Features and Options --->
General Library Settings --->
Networking Support --->
String and Stdio Support --->
Big and Tall --->
Library Installation Options --->
uClibc security related options --->
uClibc development/debugging options --->
v(+)
<Select> < Exit > < Help >

Figure 13-1. *The main screen when configuring uClibc*

The first step in configuring uClibc is to specify the type of processor that you are building it for. This not only helps determine the name of the toolchain that will be used when building uClibc, but also enables you to specify various processor-specific options. In this example, I will build uClibc for a simple i386/x86 system, but the selections for other types of processors are analogous.

To specify a particular processor, press Enter to display the Target Architecture dialog, shown in Figure 13-2.

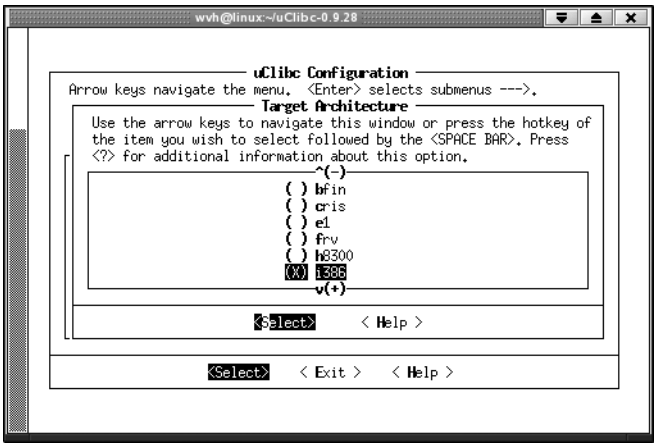

wvh@linux:~/uClibc-0.9.28
uClibc Configuration
Arrow keys navigate the menu. <Enter> selects submenus --->.
Target Architecture
Use the arrow keys to navigate this window or press the hotkey of
the item you wish to select followed by the <SPACE BAR>. Press
<?> for additional information about this option.
^(-)
( ) bfin
( ) cris
( ) e1
( ) frv
( ) h8300
(X) i386
v(+)
<Select> < Help >
<Select> < Exit > < Help >

Figure 13-2. *Specifying the target architecture when configuring uClibc*

You can use the up and down arrows or the Page Up/Page Down keys on your keyboard to scroll through the available choices on this menu. Once the correct architecture is highlighted, press the spacebar to select it. You can then use the Tab key to select the Target Architecture dialog's Exit command, which returns you to the previous menu.

After selecting the target architecture for your uClibc build, press the down-arrow key to select the Target Architecture Features and Options menu item, and press Enter to display its accompanying dialog shown in Figure 13-3.

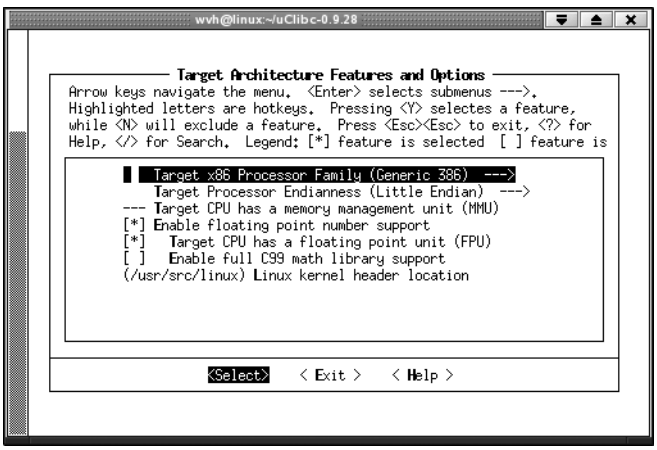

wvh@linux:~/uClibc-0.9.28
Target Architecture Features and Options
Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> selectes a feature,
while <N> will exclude a feature. Press <Esc><Esc> to exit, <?> for
Help, </> for Search. Legend: [*] feature is selected [ ] feature is
Target x86 Processor Family (Generic 386) --->
Target Processor Endianness (Little Endian) --->
--- Target CPU has a memory management unit (MMU)
[*] Enable floating point number support
[*] Target CPU has a floating point unit (FPU)
[ ] Enable full C99 math library support
(/usr/src/linux) Linux kernel header location
<Select> < Exit > < Help >

Figure 13-3. *Specifying target architecture features and options when configuring uClibc*

The Target Processor Family menu item will be highlighted. Most architectures support multiple processor families, which are groups of related processors that share instruction sets and certain functionality. To select a particular processor family, press Enter to display the Processor Family dialog shown in Figure 13-4.

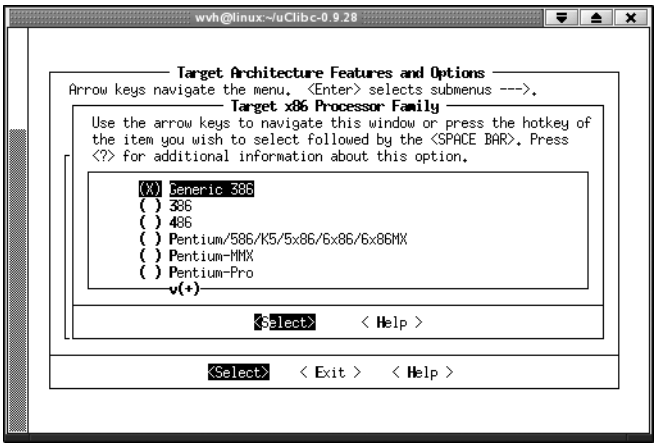

wvh@linux:~/uClibc-0.9.28
Target Architecture Features and Options
Arrow keys navigate the menu. <Enter> selects submenus --->.
Target x86 Processor Family
Use the arrow keys to navigate this window or press the hotkey of
the item you wish to select followed by the <SPACE BAR>. Press
<?> for additional information about this option.
(X) Generic 386
( ) 386
( ) 486
( ) Pentium/586/K5/5x86/6x86/6x86MX
( ) Pentium-MMX
( ) Pentium-Pro
v(+)
<Select> < Help >
<Select> < Exit > < Help >

Figure 13-4. *Specifying a particular processor family when configuring uClibc*

You can use the up and down arrows or the Page Up/Page Down keys on your keyboard to scroll through the available choices on this menu. Once the correct processor family is highlighted, press the spacebar to select it. You can then press Enter to return to the Target Architecture Features and Options dialog.

I don't want to tweak any other configuration setting at this level, so press Tab to highlight the Exit command and press Enter to redisplay the top-level configuration menu. Press the down-arrow on your keyboard to highlight the General Library Settings option, and press Enter to display the associated dialog, which is shown in Figure 13-5.

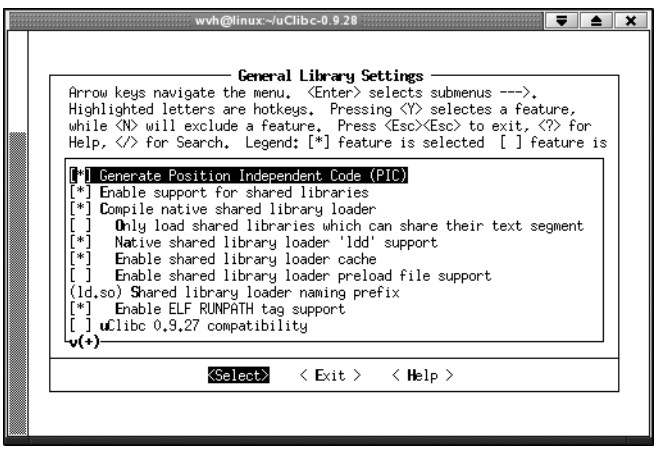

wvh@linux:~/uClibc-0.9.28
General Library Settings
Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> selectes a feature,
while <N> will exclude a feature. Press <Esc><Esc> to exit, <?> for
Help, </> for Search. Legend: [*] feature is selected [ ] feature is
[*] Generate Position Independent Code (PIC)
[*] Enable support for shared libraries
[*] Compile native shared library loader
[ ] Only load shared libraries which can share their text segment
[*] Native shared library loader 'ldd' support
[*] Enable shared library loader cache
[ ] Enable shared library loader preload file support
(ld.so) Shared library loader naming prefix
[*] Enable ELF RUNPATH tag support
[ ] uClibc 0.9.27 compatibility
v(+)
<Select> < Exit > < Help >

Figure 13-5. *Customizing library settings when configuring uClibc*

Though I am not going to change anything on this screen, this is an important screen configuration dialog to use if you are not interested in using uClibc as a shared library. Shared libraries have some advantages, as discussed earlier in this book, but many embedded developers prefer to build static applications to simplify portability. To disable shared library support in uClibc, you would use the down-arrow key to select the Enable Support for Shared Libraries menu item, and press the spacebar to deselect this option. You can always use gcc's familiar `-static` option to link statically if

shared library support is enabled, but disabling shared library support forces this behavior and can be convenient if you are working with a team of forgetful developers.

■ **Tip** Another interesting configuration item on this screen is the uClibc 0.9.27 Compatibility option. Binary compatibility is never supported, by default, between uClibc releases. This option (and, presumably, similar options in future releases of uClibc) enables you to support compatibility with the previous release, which may be important if you already have systems in the field that were compiled with this version of uClibc and you are not statically linking your applications.

Once you are finished exploring this screen and making any changes that you are interested in, press Tab to select the Exit dialog command and press Enter to redisplay the top-level uClibc configuration dialog.

Though uClibc's configuration mechanism provides other configuration screens, the last one that I want to focus on here is the Library Installation Options dialog shown in Figure 13-6. To display this dialog, use the down-arrow key to select its name and press Enter.

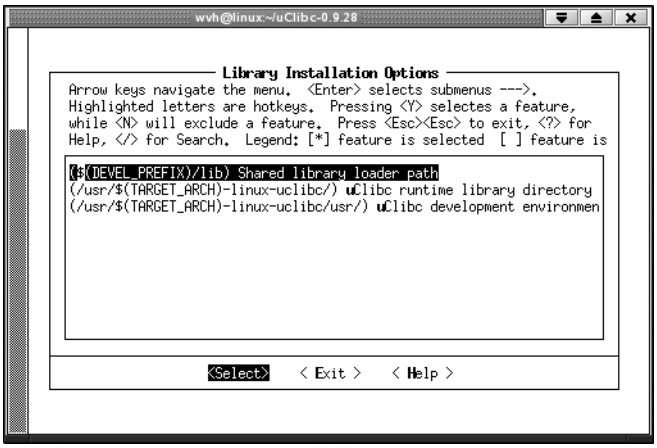

wvh@linux:~/uClibc-0.9.28
Library Installation Options
Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> selectes a feature,
while <N> will exclude a feature. Press <Esc><Esc> to exit, <?> for
Help, </> for Search. Legend: [*] feature is selected [ ] feature is
(${DEVEL_PREFIX}/lib) Shared library loader path
(/usr/$(TARGET_ARCH)-linux-uclibc/) uClibc runtime library directory
(/usr/$(TARGET_ARCH)-linux-uclibc/usr/) uClibc development environmen
<Select> < Exit > < Help >

Figure 13-6. *Customizing library installation settings when configuring uClibc*

The Library Installation Options dialog enables you to customize where applications you build on your system can expect to find the uClibc libraries, the shared library loader, and the uClibc header files and other parts of its development environment. The value shown on this screen will be the default for your system and, thus, for any applications that you link against the version of uClibc that you are building. If you are building an embedded system, you will want to copy some of the libraries from this location into any directory that represents a root filesystem for your target system and relink your applications so that they can find the target's uClibc libraries, not the ones on your development system.

For the purposes of this example, I am not going to change anything here. To exit this dialog, press the Tab key to select the dialog's Exit command and press Enter to redisplay uClibc's top-level configuration dialog.

This completes my configuration changes, so I will press the Tab key to select the Exit command and press Enter to exit from the uClibc configuration mechanism. This displays the dialog shown in Figure 13-7, which asks for confirmation that you want to save your changes.

Figure 13-7. *Saving your configuration settings when configuring uClibc*

I want to save my changes, so I will simply press Enter to do so and close the uClibc configuration application. To exit without saving your changes, you could press the Tab key to highlight No, and press Enter to exit without updating the current uClibc configuration file.

Once you have updated your uClibc configuration to reflect your changes, simply type **make** to build uClibc with those settings. The build process takes between five and ten minutes on most machines, so this may be a good time to go out for coffee.

Once the build process completes successfully, you can install your new uClibc libraries and associated files by executing the command `make install` (as root or with root privileges).

Using uClibc with gcc

As mentioned previously, the suggested and only supported way of using uClibc with gcc is to build a toolchain that works directly with uClibc. Versions of uClibc prior to 0.9.21 included a wrapper program for gcc known as gcc-uClibc (the name of the resulting binary was i386-uclibc-gcc) and an associated wrapper script called i386-uclibc-ld that convinced a version of gcc that had been built with Glibc to link with uClibc instead. This is no longer supported or distributed with any newer version of uClibc.

Of course, this is gcc, and therefore you can still convince gcc to use uClibc's include files and link with uClibc's libraries. The following is a script I used to compile and link a sample hello, world application on a 32-bit i386 test system where I had built the latest uClibc and installed it in the standard platform-specific directory `/usr/i386-linux-uclibc`. Obviously, you would ordinarily encapsulate this sort of process inside a Makefile, but it is useful to see all of the object files, libraries, and include paths required once I eliminate the standard system startup file, C library, and include directories by specifying `-nostdlib`:

```
$ gcc -nostdlib \
    -I/usr/i386-linux-uclibc/usr/include \
    -c hello.c

ld hello.o \
   /usr/i386-linux-uclibc/usr/lib/crt1.o \
   /usr/i386-linux-uclibc/usr/lib/crti.o \
   /usr/i386-linux-uclibc/usr/lib/crtn.o \
   /usr/i386-linux-uclibc/usr/lib/libc.a \
   -o hello_uclibc –static
```

When statically linked using glibc, my hello, world program on this platform was 1937152 bytes in size. When statically linked with uClibc, the same binary was 12354 bytes.

CHAPTER 14

Building and Using C Cross-Compilers

One of the most popular uses of Linux today is in developing and deploying embedded systems, which are small, typically stand-alone computer systems used throughout industry and consumer electronics for a variety of purposes. In industry, embedded Linux systems are used for everything from factory and communications infrastructure process control to monitoring and maintenance systems. In consumer electronics, embedded Linux systems are used in an incredible number of devices, including cell phones, home networking gateways and routers, wireless access points, PDAs, digital video recorders such as the TiVo, and home entertainment systems.

The primary issue in developing embedded Linux systems is bootstrapping. In order to develop applications that run on top of an embedded Linux distribution, you might think that you would need a stable version of that same embedded Linux distribution on which to compile your applications. This is a significant problem for many embedded systems, which often feature low-power processors, limited amounts of memory, and limited amounts of physical storage (and sometimes none at all). In many cases, this makes it difficult or impossible to develop embedded Linux applications directly on the system where they will finally be deployed.

Enter cross-compilation.

What Is Cross-Compilation?

Cross-compilation is a technique in which a special compiler is used that runs on a desktop development system yet produces executables that run on a different platform. This enables developers to take full advantage of the CPU horsepower, memory, and storage available on their desktop development systems (known as *host* systems in cross-compilation lingo), while still producing executables that are tailored to the architecture, instruction set, and capabilities of their embedded system (known as *target* systems in cross-compilation terms). Because the GCC compilers have been ported to so many different types of desktop development systems and have also been fine-tuned to produce binaries for so many different types of systems in general, GCC is extremely well-suited for use as a cross-compiler—everything that you need is already in there, you just have to tweak and build the compilers correctly.

This is not to say that building cross-compilers is easy. A full desktop environment for cross-compilation requires not only special versions of the compilers, configured for cross-compilation, but also requires similarly configured versions of the utilities used to build application binaries and libraries (known as binutils) and a similarly configured version of the C library that holds most of the functions required by those applications and libraries.

As I will explain throughout the remainder of this chapter, a variety of different ways to build cross-compilers are available today. The one that you choose depends on the operating system and the operating system distribution on which you are doing your development and the type of cross-compiler that you are trying to build. This chapter focuses on the most popular open source tools for

building cross-compilers—crosstool and buildroot—and also discusses how to manually build cross-compilers when necessary. Other tools for building cross-compilers are also available, such as Gentoo's crossdev project (`http://dev.gentoo.org/~redhatter/misc/xdistcc.sh.gz`) and DENX Software Engineering's ELDK (Embedded Linux Development Kit) (`http://www.denx.de/wiki/DULG/ELDK`), but are not discussed here because they are too distribution-specific or less commonly used than the other approaches discussed in this chapter.

CROSS-COMPILATION TARGETS SUPPORTED BY GCC

If you'll pardon the expression, the list of cross-compilation targets supported by GCC is a moving target, changing with each release of GCC. Rather than just listing available targets, which will certainly have changed by the time that the major release of GCC occurs, this section explains how to find the list of valid targets that can be produced from the source code for the version of the compiler that you are building.

The Wikipedia entry for GCC lists a variety of architectures and processors for which GCC is (or has been) supported. The definitive list of valid patterns for standard GCC compiler and tool prefixes is stored in the file gcc/config.gcc delivered with any GCC source code release. This file lists all of the wildcard matches that are supported in toolchain prefixes, and therefore shows you the targets that are available.

GCC prefixes are traditionally of the form `CPUTYPE-MANUFACTURER-OPERATINGSYSTEM`, though more recent GCC prefixes have adopted a new four-part form, `CPUTYPE-MANUFACTURER-KERNEL-OPERATINGSYSTEM`. For example, the standard prefix for a generic PowerPC compiler is `powerpc-unknown-linux-gnu`: a PowerPC CPU, no specific manufacturer, the Linux kernel, and a GNU operating system.

You can also examine the default prefix name that GCC's configuration utilities will try to use by executing the file config.sub in the main GCC source directory, supplying a sample prefix as an argument. For example, executing the following command shows me the prefix that the GCC tools will use if I specify a target of powerpc-linux:

```
$ ./config.sub powerpc-linux
```

```
powerpc-unknown-linux-gnu
```

This sort of test also has the advantage of displaying an "invalid configuration" message when you specify a prefix that GCC's configuration utilities do not recognize at all.

When building any cross-compiler manually, you should check the host/target–specific information in the installation notes for the version of the GCC source that you are building from. These are always located in the file doc/HTML/specific.html, relative to the root of your GCC source tree. The version of this document for the latest official release of GCC is always available at the URL `http://gcc.gnu.org/install/specific.html`. Tools such as crosstool and buildroot encapsulate this information for the versions of GCC that they can build and the targets that they support; but if you have to build a cross-compiler manually, you should always check this file for any specific information about the target platform that you are building a cross-compiler for.

Using crosstool to Build Cross-Compilers

crosstool (`http://www.kegel.com/crosstool`) is a set of shell scripts and platform- and package-specific definition files by Dan Kegel—a well-known embedded and cross-platform Linux developer and general open source advocate—that takes a significant amount of the magic out of building cross-compilers as long as you want to use a Glibc-based cross-compiler. The crosstool package was inspired by an earlier set of scripts and related information called crossgcc, which was written and maintained by Bill Gatliff (`http://www.billgatliff.com`), a well-known embedded systems developer and

consultant, but appears to have been superseded by the crosstool package. Many people have used both of these packages to build a variety of cross-compilers over the years. Nowadays crosstool is more up-to-date and better maintained and, thus, is well worth a look if you need to build your own cross-compilation toolchain.

crosstool automates the entire cross-compiler build process, automatically retrieving the source code archives for specified versions of binutils, Glibc, and GCC, extracting and configuring the contents of those archives, and building the right packages in the right order. crosstool provides an extremely flexible solution for building cross-compilers because it is driven by environment variables that are set in two configuration files:

- A *package configuration file:* Describes the versions of the packages that you want to build and retrieve, any add-on packages required by Glibc, and the version of the Linux kernel source code that you want to retrieve in order to build the headers required by Glibc and GCC.
- A *platform configuration file:* Defines the prefix used by the cross-compiler to differentiate its binaries from those for your standard desktop system, and various options to use when building packages for the specified target platform.

Driving the cross-compilation process through configuration files and environment variables makes crosstool a very flexible system that can easily be extended to support newer versions of the packages required for a cross-compiler, newer versions of the Linux kernel, and even GCC enhancements that provide additional command-line options for specific platforms. At the time this book was written, crosstool provided platform configuration files for the following platforms (unless noted, the platform configuration files have the same filename as their target processor and have a `.dat` extension):

- *alpha*: High-performance, 64-bit RISC processors originally developed by Digital Equipment Corporation (DEC) (R.I.P.) and used in many of its workstations and servers, as well as in some systems from Compaq and Hewlett-Packard. Cross-compilation toolchains produced by crosstool for this processor family have the prefix `alpha-unknown-linux-gnu`.
- *arm*: 32-bit RISC processors that support the basic ARMv4 instruction set (licensed, as all ARM instruction sets, from ARM Ltd.) and popular in small consumer electronics devices such as PDAs and cell phones. Cross-compilation toolchains produced by crosstool for this processor family have the prefix `arm-unknown-linux-gnu`.
- *arm9tdmi*: 32-bit RISC processors such as the ARM920T and ARM922T that support both the ARMv5 and 16-bit Thumb instruction sets, and are popular in consumer electronics and embedded automotive systems. Cross-compilation toolchains produced by crosstool for this processor family have the prefix `arm-9tdmi-linux-gnu`.
- *arm-iwmmxt*: 32-bit RISC processors compatible with the ARMv5 instruction set and feature instructions that support Intel Wireless MMX technology. These include processors such as Intel's PXA27x processors. Cross-compilation toolchains produced by crosstool for this processor family have the prefix `arm-iwmmxt-linux-gnu`.
- *arm-softfloat*: 32-bit RISC processors that support the basic ARMv4 instruction set and provide floating-point emulation in software. Cross-compilation toolchains produced by crosstool for this processor family have the prefix `arm-softfloat-linux-gnu`.
- *arm-xscale*: 32-bit RISC processors that are descendants of the original StrongARM instruction set which Intel acquired from DEC long ago. These processors are compatible with the ARMv5 instruction set. Cross-compilation toolchains produced by crosstool for this processor family have the prefix `arm-xscale-linux-gnu`.

- *armeb*: Big-endian 32-bit RISC processors that support the basic ARMv4 instruction set. Cross-compilation toolchains produced by crosstool for this processor family have the prefix `armeb-unknown-linux-gnu`.
- *armv5b-softfloat*: Big-endian 32-bit RISC processors that support the ARMv5 instruction set and provide floating-point emulation in software. Cross-compilation toolchains produced by crosstool for this processor family have the prefix `armv5b-softfloat-linux-gnu`.
- *i686*: Any Pentium-class IA-32 processor. Cross-compilation toolchains produced by crosstool for this processor family have the prefix `i686-unknown-linux-gnu`. crosstool provides special versions of the configuration file for this processor family that are designed to produce static cross-compilers (`i686-static.dat`) and cross-compilers that run under the Cygwin Linux emulation environment for Microsoft Windows (`i686-cygwin.dat`, using the `i686-pc-cygwin` prefix).
- *ia64*: 64-bit Intel architecture processors such as the Itanium. Cross-compilation toolchains produced by crosstool for this processor family have the prefix `ia64-unknown-linux-gnu`. Note that this target is different from the x86_64 target.
- *m68k*: Motorola 680x0 CISC processors. Cross-compilation toolchains produced by crosstool for this processor family have the prefix `m68k-unknown-linux-gnu`.
- *mips*: 32-bit big-endian MIPS processors used in many popular embedded hardware platforms. Cross-compilation toolchains produced by crosstool for this processor family have the prefix `mips-unknown-linux-gnu`.
- *mipsel*: 32-bit little-endian MIPS processors used in many popular embedded hardware platforms. Cross-compilation toolchains produced by crosstool for this processor family have the prefix `mipsel-unknown-linux-gnu`.
- *powerpc-405*: 32-bit RISC PowerPC processors. This toolchain can be used to produce binaries that will run on any PPC 4XX processors, such as the 405, 440EP, 440SP, 440GP, 440GR, 440GRx, and so on. PPC 405 cores are also used in popular FPGA PPC implementations such as the Xilinx Virtex-II Pro and Virtex-4. Cross-compilation toolchains produced by crosstool for this processor family have the prefix `powerpc-405-linux-gnu`.
- *powerpc-603*: 32-bit RISC PowerPC processors used in older Macintosh systems and various embedded computing boards. This toolchain can be used to produce binaries that will run on any PPC processors with a 603-compatible core, such as the 603, 604, 5200, 82XX, and so on. Cross-compilation toolchains produced by crosstool for this processor family have the prefix `powerpc-603-linux-gnu`.
- *powerpc-750*: 32-bit RISC PowerPC G3 processors used in many embedded computing boards and in early Apple iMacs, iBooks, and PowerBooks. This toolchain can be used to produce binaries that will run on any PPC 7XX processors, such as the 750, 750FX, 750GX, 755, and so on. Cross-compilation toolchains produced by crosstool for this processor family have the prefix `powerpc-750-linux-gnu`.
- *powerpc-860*: 32-bit RISC PowerPC PowerQUICC processors used in many embedded computing boards. This toolchain can be used to produce binaries that will run on any PowerQUICC processors, such as the 550, 860, 8XX, and so on. Cross-compilation toolchains produced by crosstool for this processor family have the prefix `powerpc-860-linux-gnu`.
- *powerpc-970*: 64-bit RISC PowerPC G5 processors used in high-end Macintosh systems, featuring an instructions set based on IBM's POWER4 architecture that is backward-compatible with the 32-bit PPC instruction sets. Cross-compilation toolchains produced by crosstool for this processor family have the prefix `powerpc-970-linux-gnu`.

- *s390*: 64-bit processors that power newer IBM mainframes, including the eServer zSeries, System z, and System z9. These systems make it quite easy to run Linux virtual machines within a vpar (virtual partition). Cross-compilation toolchains produced by crosstool for this processor family have the prefix `s390-unknown-linux-gnu`.
- *sh3*: 32-bit RISC Hitachi SuperH processors that are popular in many embedded computing boards. Cross-compilation toolchains produced by crosstool for this processor family have the prefix `sh3-unknown-linux-gnu`.
- *sh4*: 32-bit RISC Hitachi SuperH processors that are popular in many embedded computing boards. Cross-compilation toolchains produced by crosstool for this processor family have the prefix `sh4-unknown-linux-gnu`.
- *sparc*: 32-bit RISC processors originally developed by Sun Microsystems and licensed to other manufacturers for use in a variety of workstations and embedded computing boards. (The latter were especially popular in the telecommunications industry.) Cross-compilation toolchains produced by crosstool for this processor family have the prefix `sparc-unknown-linux-gnu`.
- *sparc64*: 64-bit RISC processors developed by Sun Microsystems and primarily used in high-end Sun workstations and servers. Cross-compilation toolchains produced by crosstool for this processor family have the prefix `sparc64-unknown-linux-gnu`.
- *x86_64*: 64-bit processors from manufacturers such as AMD, including Intel's latest Pentium 4 processors. The instruction sets for these processors are compatible with the IA-32 Intel instruction set but not with the IA-64 instruction set. Cross-compilation toolchains produced by crosstool for this processor family have the prefix `x86_64-unknown-linux-gnu`.

Though crosstool provides platform configuration files for all of these platforms, different versions of the packages used to create the cross-compilers require different sets of patches, and (unfortunately) not all combinations of the binutils, Glibc, and GCC packages will successfully compile using crosstool. Luckily, the crosstool site provides a matrix of platforms and package versions, indicating which versions build correctly with what combinations of packages. The page for crosstool version 0.42 is `http://www.kegel.com/crosstool/crosstool-0.42/buildlogs`. This page is generated automatically during a nightly build process that tries all of the specified combinations; it can be extremely useful in identifying a set of packages that will build a valid cross-compiler for your target platform. If you need a specific combination of package versions that do not work together under crosstool, you may have to create your own package configuration file (for newer, not-yet-supported package and Linux kernel versions), or manually build your own cross-compiler after patching the source for the appropriate package, as described later in this chapter in the section titled "Building Cross-Compilers Manually."

In addition to the environment variables defined in the configuration files, you must also set three environment variables manually within the shell where you are executing the crosstool script. These are the following:

- `GCC_LANGUAGES`: A string containing a comma-separated list of the languages for which you want to build cross-compilers. These are the same values that you can pass to a standard GCC build using the `--enable-languages` option for a specific version of the GCC compilers.
- `RESULT_TOP`: The root of the directory tree below which you want to install the finished cross-compilers, associated tools, libraries, and include files.
- `TARBALLS_DIR`: The name of a temporary directory in which crosstool will save the source code archives that it retrieves for you during the cross-compiler build process. Saving these files locally can reduce build time if you are building multiple cross-compilers that each use the same versions of binutils, Glibc and related packages, or GCC.

The next two sections describe how to retrieve and use the crosstool package to build an appropriate cross-compiler for your embedded target platform.

Retrieving the crosstool Package

You can always retrieve the latest version of crosstool from the Downloads section of the crosstool site at `http://www.kegel.com/crosstool` to see if any newer versions are available when using crosstool—new versions are almost guaranteed to be better.

Once you retrieve the crosstool tarball, you can extract its contents using a standard tar command such as `tar zxvf cross-tool-0.42.tgz` (your filename may be different). This will create a directory with the name of the crosstool script and its current version number, which in this case would be a directory named `crosstool-0.42`. Change directory to this directory, and you're ready to go.

Building a Default Cross-Compiler Using crosstool

In a nutshell, building your cross-compiler requires that you set three environment variables described in the introduction to this section; identify the package and platform configuration files that are appropriate for your target; and then execute the all.sh script, using the selected configuration files to set the standard environment variables for your packages and target platform.

■ **Note** This section discusses building a cross-compiler with one of the combinations of binutils, GCC, and Linux kernel headers that are defined in one of the configuration files provided as part of the crosstool package. If you need to build a cross-compiler using some other combination of these packages and headers, you will have to create your own configuration file(s) and patch sets. For information about doing this, see the next section, "Building a Custom Cross-Compiler Using crosstool."

For example, to build a cross-compiler for the arm-xscale target using crosstool, do the following:

1. Make sure that you're working in the directory created when you extracted the crosstool tarball. At the time that this book was written, the directory was `crosstool-0.42`.
2. Check the Web page at `http://www.kegel.com/crosstool/crosstool-0.42/buildlogs` to determine which combination of packages will successfully build a cross-compiler for your target platform. A completely green square indicates that the combination of tools packages identified in the column header will build successfully, and once built, it can build a Linux kernel and root filesystem packages. You can also use crosstool to build cross-compilers using combinations of packages and headers that are not officially supported by crosstool, but you're on your own for patches, bug fixes, and so on. For information about doing this, see the next section, "Building a Custom Cross-Compiler Using crosstool."
3. Create the directory in which you want to install your cross-compiler (if it does not already exist) and make sure that you have write permission there.
4. Set the environment variable `RESULT_TOP` to the name of the directory in which you want to install the cross-compiler that you are building, using a command such as the following:

```
$ export RESULT_TOP=/tools/cross-compilers
```

■ **Tip** If you are building cross-compilers that you want to share with multiple users, you may want to build and install them into an NFS-mounted directory that is mounted at the same place on multiple machines. For example, in my case, `/tools` is an NFS-mounted root filesystem that is mounted on all of my systems. Alternately, if you want the toolchains that you are building to be statically linked in order to simplify copying them to different systems, you can set the following environment variables as shown before beginning the build process:

```
$ export BINUTILS_EXTRA_CONFIG="LDFLAGS=-all-static"
$ export GCC_EXTRA_CONFIG="LDFLAGS=-static"
```

5. Set the environment variable `GCC_LANGUAGES` to the list of languages for which you want to build cross-compilers, as in the following example:

   ```
   $ export GCC_LANGUAGES="c,c++,java"
   ```

6. Create a directory to hold the packages and patches that the crosstool package will download, if one does not already exist. Set the environment variable `TARBALLS_DIR` to the name of this directory, as in the following example:

   ```
   $ mkdir /tmp/crosstool-download
   $ export TARBALLS_DIR=/tmp/crosstool-download
   ```

7. Start the crosstool process using the appropriate package and platform configuration files, using a command such as the following:

   ```
   $ eval 'cat arm-xscale.dat gcc-3.4.5-glibc-2.3.6.dat' all.sh –notest
   ```

   This command line will build an arm-xscale cross-compiler that uses gcc 3.4.5 and Glibc 2.3.6.

At this point, the crosstool script will begin to retrieve all of the packages required for the specified cross-compiler, build them in order, and then install the final results in subdirectories of `/tools/cross-compilers`. These subdirectories will be named after the package versions used to build the cross-compilers, with the platform names located below them. In this example, the arm-xscale cross-compiler's `bin`, `lib`, `include`, `share`, and other directories will actually be located in the directory `/tools/cross-compilers/gcc-3.4.5-glibc-2.3.6/arm-xscale-linux-gnu`.

■ **Tip** The crosstool package comes with a number of scripts with names of the form *platform*_demo.sh, such as demo_arm.sh, demo-armeb.sh, demo-m68k.sh, demo-mips.sh, and so on. These scripts contain predefined settings for all of the environment variables and configuration files required to build a working cross-compiler for the specified platform, based on successful runs from the crosstool project's nightly builds. If you are comfortable with the settings in the script and have no specific package version or Linux kernel header preferences, you can just run one of these to build a working cross-compiler for your target platform.

Building a Custom Cross-Compiler Using crosstool

If you need to use specific versions of the binutils, GCC, Glibc (and add-ons), or the Linux kernel headers, you can create your own package and platform configuration files for crosstool and see if a cross-compiler using that combination will build successfully in the crosstool infrastructure.

If you simply want to attempt to build a cross-compiler with a newer version of one of its component software packages or a newer version of the Linux kernel headers, the only configuration file that you will have to update is the package configuration file. The contents of a sample package configuration file, in this case the file gcc-3.4.5-glibc-2.3.6.dat, are the following:

```
BINUTILS_DIR=binutils-2.15
GCC_DIR=gcc-3.4.5
GLIBC_DIR=glibc-2.3.6
LINUX_DIR=linux-2.6.8
LINUX_SANITIZED_HEADER_DIR=linux-libc-headers-2.6.12.0
GLIBCTHREADS_FILENAME=glibc-linuxthreads-2.3.6
```

To substitute a newer version of any of these packages, assuming that it can be downloaded from the same site as previous versions, you must just copy an existing package configuration file to a name that reflects the new package(s), and then modify the new package configuration file to reflect the new package(s). For example, suppose that you wanted to build a version of the arm-xscale cross-compiler used as an example in the previous section using GCC 4.2.0. You would first copy a file such as gcc-3.4.5-glibc-2.3.6.dat to an appropriate name such as gcc-4.2.0-glibc-2.3.6.dat, and then modify that file to set the `GCC_DIR` environment variable to the correct internal name for GCC 4.2.0, as in the following example:

```
BINUTILS_DIR=binutils-2.15
GCC_DIR=gcc-4.2.0
GLIBC_DIR=glibc-2.3.6
LINUX_DIR=linux-2.6.8
LINUX_SANITIZED_HEADER_DIR=linux-libc-headers-2.6.12.0
GLIBCTHREADS_FILENAME=glibc-linuxthreads-2.3.6
```

If you have patches that you want to apply to any of the newer components that you are downloading and building, you will need to create directories with the base names of those under the patches directory in your crosstool installation directory, and then put each patch in the right directory. For example, assume that I had a patch named `arm-fixes.patch` that I needed to apply to the version of GCC 4.2 that I specified in my previous changes to my sample package configuration file. If the directory `patches/gcc-4.2.0` did not already exist in my cross-tool installation directory, I would need to create that directory and then copy my patch there. The crosstool scripts will automatically identify and apply it to the source code that they have retrieved for GCC 4.2.0 before building GCC 4.2.0.

If this is the only change that you want to make, you could now re-run the crosstool scripts as described in the previous section, specifying the name of your new package configuration file on the command line instead of the one that you previously used.

You can also try to build other types of cross-compilers than the basic ones that are supported by crosstool by creating your own platform configuration file. The contents of a sample platform configuration file, in this case the file arm-xscale.dat, are the following:

```
KERNELCONFIG=`pwd`/arm.config
TARGET=arm-xscale-linux-gnu
TARGET_CFLAGS="-O"
GCC_EXTRA_CONFIG="--with-cpu=xscale  --enable-cxx-flags=-mcpu=xscale"
```

To attempt to use crosstool to build a new type of cross-compiler, you would create a platform configuration file and set these environment variables appropriately for the architecture and processor targeted by the cross-compiler that you are trying to build. You will also need to copy a sample kernel configuration file for the new architecture and processor, consistent with the version of the Linux kernel headers names in the package configuration file that you are using, to the file identified by the `KERNELCONFIG` environment variable. You would then re-run the all.sh script as described in the

previous section, specifying the new platform configuration file on the command-line instead of the one that you previously used, and hope for the best.

For additional details about using and customizing crosstool, see the official crosstool-howto at `http://www.kegel.com/crosstool` (search for the link to the how-to file).The URL will differ if you are using a newer version of crosstool, but this how-to (or an updated version) is the definitive source for information on how to use, customize, and extend crosstool.

■ **Tip** If you create new package or platform configuration files and can successfully build a cross-compiler using them, please contribute them back to the crosstool project so that everyone can benefit from your efforts. Thanks in advance.

Using buildroot to Build uClibc Cross-Compilers

The buildroot project (`http://buildroot.uclibc.org`) is a project created by the patron saint of embedded Linux computing, Erik Anderson. The buildroot project simplifies the process of building uClibc-based cross-compilation toolchains (`http://www.uclibc.org`) and using them to create basic root filesystems that use BusyBox (`http://www.busybox.net`). The uClibc alternate C library is discussed in Chapter 13. A complete discussion of the BusyBox project, a multicall binary that provides all of the functionality required for a basic Linux root filesystem and which is commonly used in embedded system development projects, is outside the scope of this book.

At the time this book was written, buildroot supported building cross-compilers, BusyBox, and a BusyBox-based root filesystem for the following platforms:

- *alpha*: High-performance 64-bit RISC processors originally developed by Digital Equipment Corporation (DEC) and used in many of its workstations and servers, as well as in some systems from Compaq and Hewlett-Packard.
- *arm*: 32-bit RISC processors that support various instruction sets licensed from ARM Ltd. Specific versions of the ARM processor that are supported by buildroot are generic_arm (using only the ARMv4 instruction set), ARM610, ARM710, ARM720T, ARM920T, ARM922T, ARM926T, ARM1136jf_s, AA110, AA1100, and XScale.
- *armeb*: 32-bit RISC processors in big-endian mode that support various instruction sets licensed from ARM. The same processor versions supported for the ARM architecture are supported for the armeb architecture.
- *cris*: The Code Reduced Instruction Set processors available from Axis Solutions and frequently used in embedded networking hardware.
- *i386*: 32-bit processors from Intel and others that support the IA-32 instruction set. Specific versions of the i386 processor that are supported by buildroot are i686 (the default), i586, i486, and i386.
- *m68k*: Motorola 680x0 CISC processors.
- *mips*: 32-bit big-endian MIPS processors used in many popular embedded hardware platforms.
- *mipsel*: 32-bit little-endian MIPS processors used in many popular embedded hardware platforms.
- *nios2*: The Nios II processors from Altera that are optimized for use in FPGAs.
- *powerpc*: 32-bit RISC PowerPC processors.

- *sh*: 32-bit RISC Hitachi SuperH processors that are popular in many embedded computing boards. Specific versions of the SH processor that are supported by `buildroot` are SH2a_nofpueb, SH2eb, SH3, SH3eb, SH4 (the default), and SH4eb.
- *sparc*: 32-bit RISC processors originally developed by Sun Microsystems and licensed to other manufacturers for use in a variety of workstations and embedded computing boards.
- x86_64: 64-bit processors from manufacturers such as AMD. The instruction sets for these processors are compatible with the IA-32 Intel instruction set but not with the IA-64 instruction set.

The buildroot project unifies the GCC, uClibc, and BusyBox projects into a single, coherent configuration and automatic build process that features a unified terminal-oriented configuration environment that is similar to the configuration mechanisms used by stand-alone BusyBox and uClibc builds. The following sections discuss how to retrieve and install buildroot and how to use buildroot to create working cross-compilers for a variety of target architectures. The discussion of using buildroot explicitly avoids a discussion of configuring BusyBox and building root filesystems; as you'll see in the section "Building a Cross-Compiler Using buildroot," one of the steps in the configuration process I will use is to disable building BusyBox or a root filesystem image.

Retrieving the buildroot Package

The source code for the buildroot package is available in two basic ways: by downloading a daily snapshot of the buildroot source code, or by directly retrieving a local copy of the actual buildroot source code tree. You can always retrieve a daily snapshot of the source code from `http://buildroot.uclibc.org/downloads/snapshots`, where each day's snapshot has a name of the form buildroot-YYYYMMDD.tar.gz, where *YYYY*, *MM*, and *DD* represent the four-digit year, the two-digit month, and the two-digit day for which you want to retrieve the source code snapshot. Once the archive of the daily snapshot downloads to your system, you can extract its contents using a standard command such as `tar zxvf` *`filename`*.

Retrieving a local copy of the source code for the buildroot source code archive is potentially more up-to-date and much more fun than downloading a tarball, so I will focus on that approach in this section.

The buildroot source tree is archived and accessible through the use of the Subversion Source Code Control System (SCCS). Subversion (`http://subversion.tigris.org`) is a powerful open source, network-enabled, and WebDAV-based (Web-based Distributed Authoring and Versioning) SCCS that enables you to easily and recursively retrieve the up-to-date source code for buildroot directly from its main source code repository over the Internet. You will need to have the Subversion client, svn, installed on your system in order to retrieve the buildroot source code tree over the Internet. On systems that manage installed software through the use of a package manager, the svn client is part of the standard subversion package and is typically installed as /usr/bin/svn. The Subversion source code and precompiled packages for use with many different types of systems are available from the Subversion Web site at `http://subversion.tigris.org/project_packages.html`.

Assuming that you want to retrieve buildroot using the svn client, you simply execute the following command in the directory where you want a local copy of the buildroot source code tree to be created:

```
$ svn co svn://uclibc.org/trunk/buildroot
```

After a short wait, you will see the svn command retrieve each of the files in the buildroot source tree, creating any necessary directories or subdirectories. Once this command completes, you're ready to get started!

■ **Tip** Should you subsequently want to synchronize your local copy of the buildroot source code, retrieving any updates that are available, you can do so by simply executing the following command from that same location:

```
$ svn update
```

Building a Cross-Compiler Using buildroot

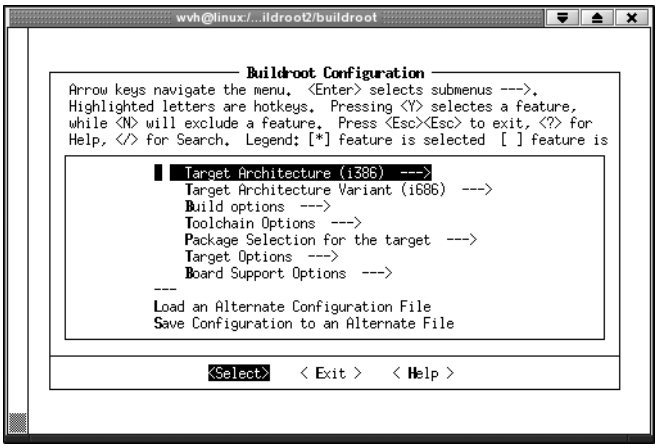
As we will see in this section, building a uClibc-based cross-compiler using buildroot is quite easy to do.

Create the directory in which you want to install your cross-compiler (if it does not already exist) and make sure that you have write permission there. By default, buildroot builds its cross-compilation toolchains for internal use by the root filesystem build process. If you want to build a stand-alone toolchain (as I do in the following example), you'll want to install it in a location in your development environment to which you (and probably others) have easy access. Throughout this example, I will use the directory `/tools/cross-compilers` as a generic location for installing the target toolchain. Unlike toolchains built with crosstool which add their own subdirectories, toolchains built with buildroot install directly into a specified directory, so I will specify an architecture/processor–specific subdirectory when the time comes in the buildroot configuration process.

■ **Tip** If you are building cross-compilers that you want to share with multiple users, you may want to build and install them into an NFS-mounted directory that is mounted at the same place on multiple machines. For example, in my case, `/tools` is an NFS-mounted root filesystem that is mounted on all of my systems.

To use buildroot to build a cross-compiler that uses uClibc, do the following:

1. Change directory to the buildroot directory created when you checked out the buildroot source code tree or extracted the contents of a daily archive snapshot.
2. Execute the `make menuconfig` command to begin the configuration process. After buildroot compiles a few binaries required to support the configuration interface, a screen like the one shown in Figure 14-1 displays.


wvh@linux:/...ildroot2/buildroot
Buildroot Configuration
Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> selectes a feature,
while <N> will exclude a feature. Press <Esc><Esc> to exit, <?> for
Help, </> for Search. Legend: [*] feature is selected [ ] feature is
Target Architecture (i386) --->
Target Architecture Variant (i686) --->
Build options --->
Toolchain Options --->
Package Selection for the target --->
Target Options --->
Board Support Options --->

Load an Alternate Configuration File
Save Configuration to an Alternate File
<Select> < Exit > < Help >

Figure 14-1. *buildroot's initial configuration screen*

3. When you first display the buildroot configuration interface, the Target Architecture menu item is highlighted. Press Enter to display the submenu that enables you to select the target architecture for which you want to build a cross-compilation toolchain. A screen like the one shown in Figure 14-2 displays.

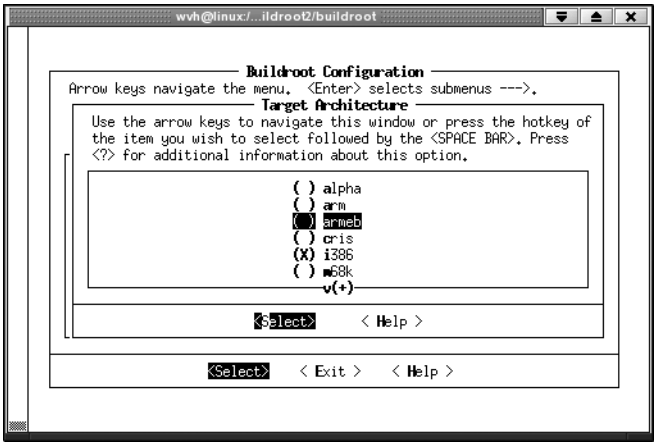


Figure 14-2. *Selecting the target architecture for your toolchain*

4. Use the up/down arrow keys on your keyboard to select the target platform for which you want to create a cross-compiler. In this example, I will build an armeb toolchain, so I've selected *armeb*. Press the spacebar once the proper target is highlighted. That target is selected and the top configuration screen (shown in Figure 14-1) redisplays with the target that you selected displayed in the Target Architecture menu item.

■ **Note** Depending on the target architecture you have selected, you may see a Target Architecture Variant menu on the screen, as shown in Figure 14-1. buildroot builds a single toolchain for some processor families, such as the PowerPC processors, so this option may no longer be present, even if it was present when you started your buildroot configuration (because there are multiple architecture variants available for the default i386 platform).

---

5. If a Target Architecture Variant configuration option is present, use the up/down arrow keys on your keyboard to scroll down to that configuration item, and press Enter. A Target Architecture Variant screen displays, as shown in Figure 14-3. The contents of this screen will differ depending on your original Target Architecture selection. The example shown in Figure 14-3 is for the armeb processor. Use the up/down arrow keys on your keyboard to scroll down to the appropriate variant for your target architecture. I have highlighted the ARM920T architecture variant since that is the processor used in my target platform for this example. Once you have selected the appropriate processor for your target platform, press the spacebar and redisplay the top-level buildroot configuration screen, with the target that you selected displayed in the Target Architecture Variant menu item.

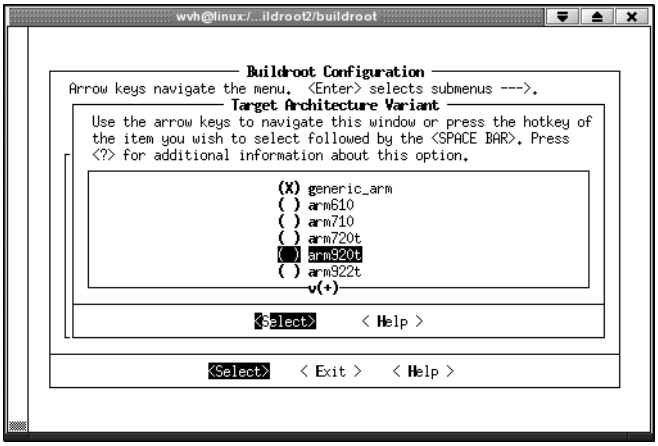

wvh@linux:/...ildroot2/buildroot
Buildroot Configuration
Arrow keys navigate the menu. <Enter> selects submenus --->.
Target Architecture Variant
Use the arrow keys to navigate this window or press the hotkey of the item you wish to select followed by the <SPACE BAR>. Press <?> for additional information about this option.
(X) generic_arm
( ) arm610
( ) arm710
( ) arm720t
( ) arm920t
( ) arm922t
v(+)
<Select> < Help >
<Select> < Exit > < Help >

Figure 14-3. *Optionally specifying architecture variants*

6. Use the up/down arrow keys on your keyboard to scroll down to the Build Options configuration item, and press Enter. The Build Options screen displays, as shown in Figure 14-4.

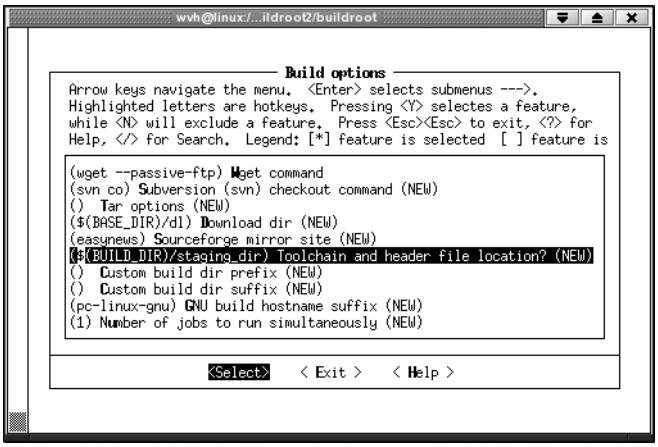

wvh@linux:/...ildroot2/buildroot
Build options
Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> selectes a feature, while <N> will exclude a feature. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] feature is selected [ ] feature is
(wget --passive-ftp) Wget command
(svn co) Subversion (svn) checkout command (NEW)
() Tar options (NEW)
($(BASE_DIR)/dl) Download dir (NEW)
(easynews) Sourceforge mirror site (NEW)
($(BUILD_DIR)/staging_dir) Toolchain and header file location? (NEW)
() Custom build dir prefix (NEW)
() Custom build dir suffix (NEW)
(pc-linux-gnu) GNU build hostname suffix (NEW)
(1) Number of jobs to run simultaneously (NEW)
<Select> < Exit > < Help >

Figure 14-4. *Changing build options for your toolchain*

7. Use the up/down arrow keys on your keyboard to move the cursor to the Toolchain and Header File Location entry (shown highlighted in Figure 14-4), and press Enter to display a dialog that enables you to change the value to reflect where you want to install the cross-compiler that you are building. Toolchains built with buildroot install directly into a specified directory, so you will want to specify an architecture/processor–specific target directory. Figure 14-5 shows this dialog after entering the new value `/tools/cross-compilers/armeb-arm920t`. Press Enter to close this dialog and press the Escape key to return to the main buildroot configuration screen.

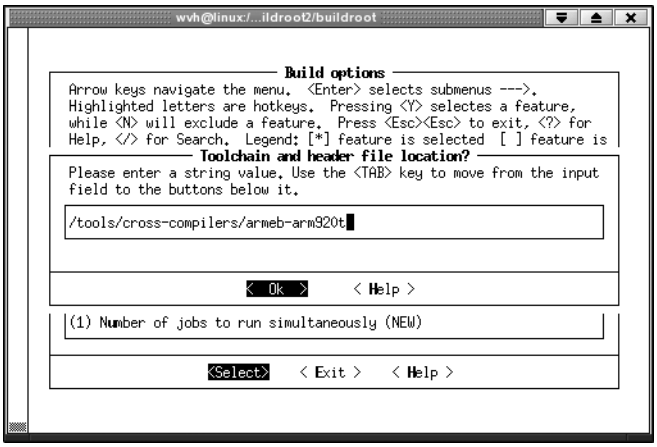

wvh@linux:/...ildroot2/buildroot
Build options
Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> selectes a feature,
while <N> will exclude a feature. Press <Esc><Esc> to exit, <?> for
Help, </> for Search. Legend: [*] feature is selected [ ] feature is
Toolchain and header file location?
Please enter a string value. Use the <TAB> key to move from the input
field to the buttons below it.
/tools/cross-compilers/armeb-arm920t
< Ok >
< Help >
(1) Number of jobs to run simultaneously (NEW)
<Select>
< Exit >
< Help >

Figure 14-5. *Identifying the install location for your toolchain*

8. Use the up/down arrow keys on your keyboard to move the cursor to the Toolchain Options menu item, and press Enter to display the Toolchain Options configuration screen, as shown in Figure 14-6.

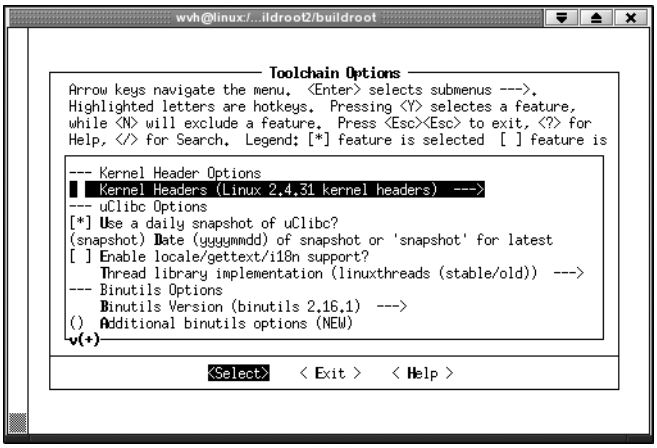

wvh@linux:/...ildroot2/buildroot
Toolchain Options
Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> selectes a feature,
while <N> will exclude a feature. Press <Esc><Esc> to exit, <?> for
Help, </> for Search. Legend: [*] feature is selected [ ] feature is
--- Kernel Header Options
Kernel Headers (Linux 2.4.31 kernel headers) --->
--- uClibc Options
[*] Use a daily snapshot of uClibc?
(snapshot) Date (yyyymmdd) of snapshot or 'snapshot' for latest
[ ] Enable locale/gettext/i18n support?
Thread library implementation (linuxthreads (stable/old)) --->
--- Binutils Options
Binutils Version (binutils 2.16.1) --->
() Additional binutils options (NEW)
v(+)
<Select>
< Exit >
< Help >

Figure 14-6. *Setting general toolchain options*

9. By default, buildroot uses headers for the latest 2.4-based Linux kernel. Though stable, this is hardly very interesting nowadays. If you don't want to change the version of the Linux kernel headers used when building your toolchain for some reason, skip to step 11. To use a newer set of Linux kernel headers (required for the uClibc and GCC build processes), use the arrow keys to move the cursor to the Kernel Headers entry (shown in Figure 14-6), and press Enter to display a dialog that enables you to select a newer set of kernel headers, as shown in Figure 14-7.
10. To change the version of the Linux kernel headers used in your toolchain build process, use the up/down arrow keys to scroll down until the Linux 2.6.12 kernel headers menu entry is selected, and press Enter to select this item and return to the Toolchain Options screen. Press Escape to return to the top level of the buildroot configuration screens.

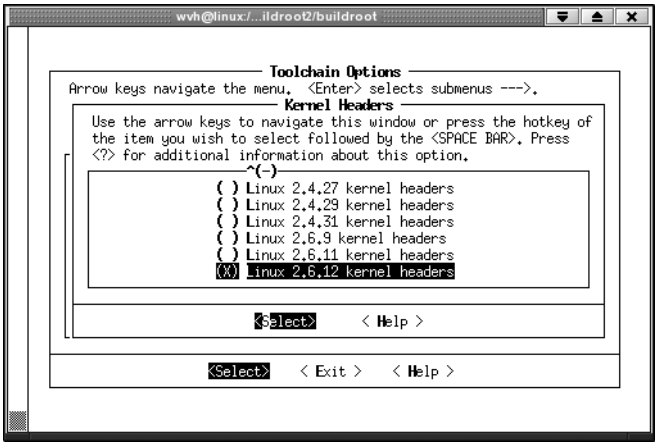

wvh@linux:/...ildroot2/buildroot
Toolchain Options
Arrow keys navigate the menu. <Enter> selects submenus --->.
Kernel Headers
Use the arrow keys to navigate this window or press the hotkey of the item you wish to select followed by the <SPACE BAR>. Press <?> for additional information about this option.
( ) Linux 2.4.27 kernel headers
( ) Linux 2.4.29 kernel headers
( ) Linux 2.4.31 kernel headers
( ) Linux 2.6.9 kernel headers
( ) Linux 2.6.11 kernel headers
(X) Linux 2.6.12 kernel headers
<Select> < Help >
<Select> < Exit > < Help >

Figure 14-7. *Identifying the kernel headers used when building your toolchain*

Note The configuration screen shown in Figure 14-6 provides many options beyond simply customizing Linux kernel header versions. Other configuration options on this screen enable you to configure things such as internationalization support, GCC options such as multilib support, general options such as large file support, the versions of the binutils, GCC, and GDB packages that you are building for your toolchain, and so on. If you need to enable or disable such features or need to produce a toolchain that uses a specific version of any of these packages, you should explore the other options on the configuration screen shown in Figure 14-6.

---

11. Use the up/down arrow keys to select the Package Selection for Target configuration option displays, and press Enter to display the configuration dialog shown in Figure 14-8. Since we are only interested in building a cross-toolchain, use the arrow keys to scroll down until the BusyBox item is selected.

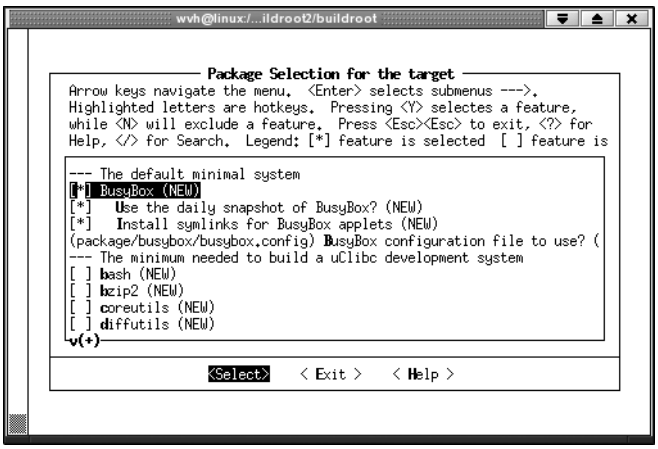

wvh@linux:/...ildroot2/buildroot
Package Selection for the target
Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> selectes a feature, while <N> will exclude a feature. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] feature is selected [ ] feature is
--- The default minimal system
[*] BusyBox (NEW)
[*] Use the daily snapshot of BusyBox? (NEW)
[*] Install symlinks for BusyBox applets (NEW)
(package/busybox/busybox.config) BusyBox configuration file to use? (
--- The minimum needed to build a uClibc development system
[ ] bash (NEW)
[ ] bzip2 (NEW)
[ ] coreutils (NEW)
[ ] diffutils (NEW)
v(+)
<Select> < Exit > < Help >

Figure 14-8. *Selecting packages to build along with the toolchain*

12. Press the spacebar to deselect building BusyBox. The screen changes to look like that shown in Figure 14-9, illustrating the fact that BusyBox and a root filesystem based on it will no longer be built as part of your toolchain build.

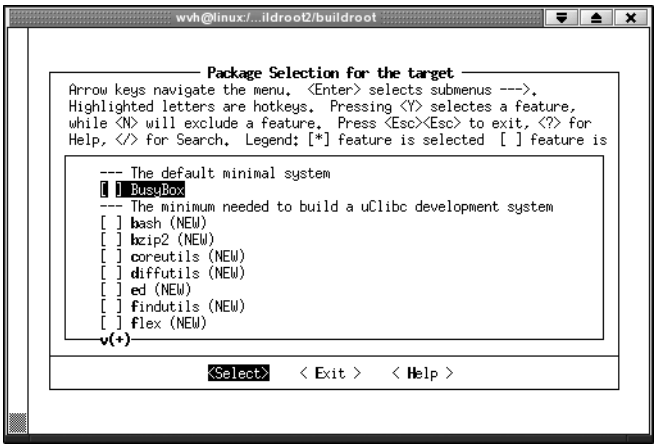

wvh@linux:/...ildroot2/buildroot
Package Selection for the target
Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> selectes a feature,
while <N> will exclude a feature. Press <Esc><Esc> to exit, <?> for
Help, </> for Search. Legend: [*] feature is selected [ ] feature is
--- The default minimal system
[ ] BusyBox
--- The minimum needed to build a uClibc development system
[ ] bash (NEW)
[ ] bzip2 (NEW)
[ ] coreutils (NEW)
[ ] diffutils (NEW)
[ ] ed (NEW)
[ ] findutils (NEW)
[ ] flex (NEW)
v(+)
<Select> < Exit > < Help >

Figure 14-9. *Disabling the BusyBox build*

13. Use the up/down arrow keys to select the Target Options menu item, and press Enter to display the configuration screen shown in Figure 14-10. Use the up/down arrow keys to select the ext2 root filesystem option, and press the spacebar to deselect this option. This will prevent the buildroot compilation process from attempting to build an empty ext2 filesystem. The Target Options configuration screen will update to reflect the fact that no filesystems will be created during the build process for your toolchain. Press Escape to redisplay the top-level buildroot configuration screen.

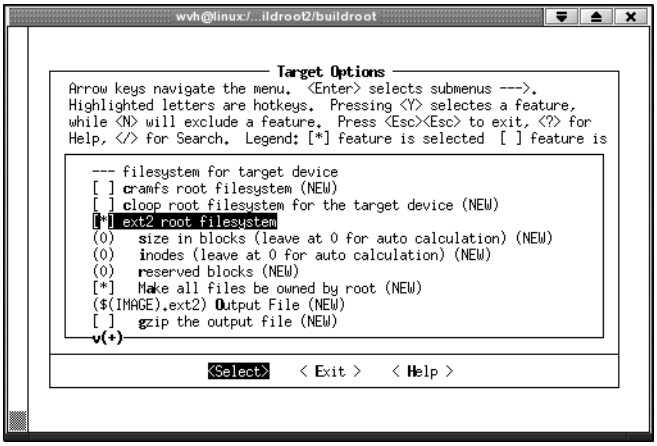

wvh@linux:/...ildroot2/buildroot
Target Options
Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> selectes a feature,
while <N> will exclude a feature. Press <Esc><Esc> to exit, <?> for
Help, </> for Search. Legend: [*] feature is selected [ ] feature is
--- filesystem for target device
[ ] cramfs root filesystem (NEW)
[ ] cloop root filesystem for the target device (NEW)
[*] ext2 root filesystem
(0) size in blocks (leave at 0 for auto calculation) (NEW)
(0) inodes (leave at 0 for auto calculation) (NEW)
(0) reserved blocks (NEW)
[*] Make all files be owned by root (NEW)
($(IMAGE).ext2) Output File (NEW)
[ ] gzip the output file (NEW)
v(+)
<Select> < Exit > < Help >

Figure 14-10. *Disabling ext2 filesystem creation during the toolchain build*

14. Press the Tab key to select the Exit command on the top-level buildroot configuration screen. The screen shown in Figure 14-11 displays.

Figure 14-11. *Saving your configuration changes*

15. Press Enter to save your buildroot configuration changes and exit the buildroot configuration screens. At this point, you can simply type **make** and wait for buildroot to begin retrieving the files required for a cross-compilation toolchain for your selected target. After retrieving the kernel and the uClibc source code, the make process will automatically start the command-line uClibc configuration editor for you, as shown in Figure 14-12.

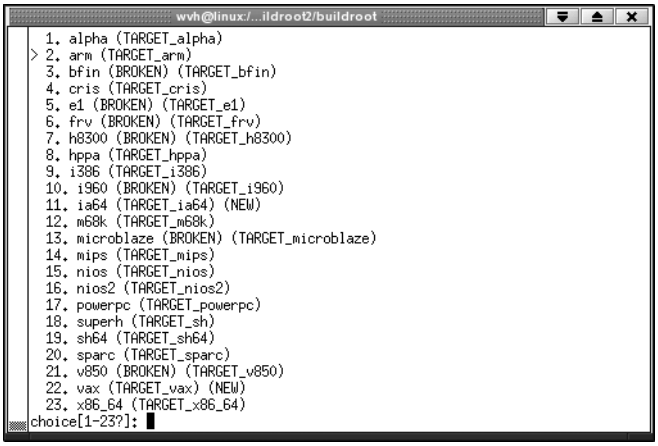


Figure 14-12. *Fine-tuning uClibc build settings for your target architecture*

16. A greater-than sign (>) displays at the right side of the uClibc configuration editor, highlighting the architecture that it believes is your target architecture. If the indicated architecture is correct (No. 2 in Figure 14-12), press Enter to accept that value and continue with the uClibc configuration process.

17. Next, depending on your target architecture, the configuration process will ask a series of architecture and process-specific questions, such as whether you want to support a specific application binary interface, as shown in Figure 14-13, or make more refined selections regarding the type of processor for which you are building the toolchain, as shown in Figure 14-14. If a numeric selection is required, a suggested answer is always highlighted with a greater-than sign, as in Figure 14-12. Most of these configuration questions are yes or no questions to which you can simply answer y or n, or press Enter to accept a default answer. The uppercase value inside square brackets at the right of each question is the default value, so if that value is appropriate, you can simply press Enter to continue.

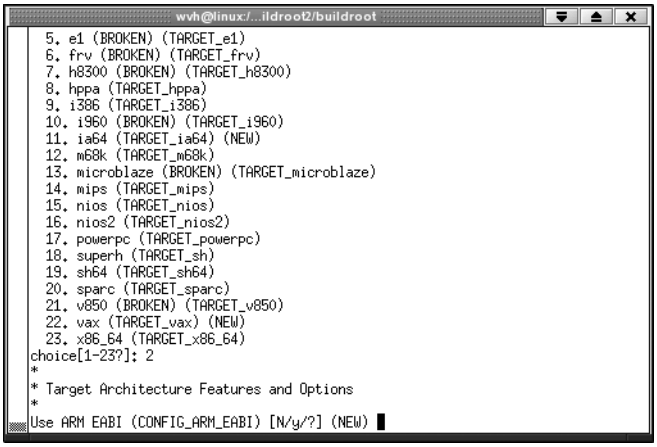


Figure 14-13. *Fine-tuning other uClibc build settings for your cross-compiler*

■ **Tip** When building your first cross-compiler using buildroot, simply press Enter to supply the default response to each detailed uClibc configuration question after verifying your target architecture. Once you have verified that you can successfully create a toolchain that will produce binaries for your target system, you can always go back and change the values that you supplied by rerunning the make command again.

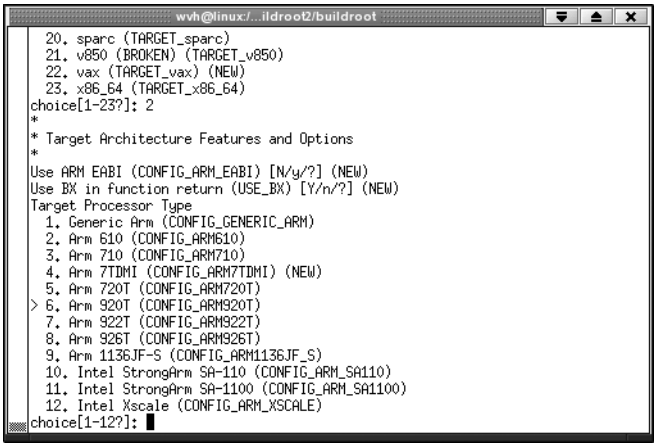


Figure 14-14. *Processor-specific build settings for your cross-compiler*

The remainder of the buildroot cross-compilation process doesn't require any manual intervention. Building a toolchain can take a significant amount of time, depending on the speed and memory available on your build system. When the build process completes successfully, your shell prompt will redisplay in the window from which you initiated the buildroot configuration process. If you see that message, congratulations—you've built a cross-compiler! If you've encountered problems, don't give up hope—see the next section for suggestions on debugging and resolving problems building a toolchain using buildroot.

Debugging and Resolving Toolchain Build Problems in buildroot

If you are reading this section, you have either encountered a problem building a uClibc-based toolchain using buildroot or you are simply curious about problems that you might encounter.

Frankly, the most common source of problems building toolchains using the buildroot package at this point is the set of architecture/processor–specific default values provided during the uClibc configuration process. In the previous section, I suggested that you initially accept the defaults, which are usually correct and therefore typically work. Unfortunately, build infrastructures such as buildroot can't anticipate all possible conflicts between the settings used by binutils, GCC, uClibc, and the Linux kernel headers that you've selected. That would be nice, but so would a bush that grows candy canes.

Though I suggest that you always try accepting as many default values as possible during the buildroot and uClibc configuration processes, you will still occasionally have to manually reset some of them. For example, when building the ARM920T toolchain (in the previous section) I accepted the default suggestion that it should be able to use the BX instruction to enter/leave ARM Thumb mode, as shown in Figure 14-15. Unfortunately, in certain cases, this instruction isn't supported for my selected processor by the version of the GNU assembler that I am using, and I see the following error during the build process:

```
  CC ldso/ldso/ldso.oS
/tmp/ccwzLCjF.s: Assembler messages:
/tmp/ccwzLCjF.s:39: Error: selected processor does not support `bx r6'
make[1]: *** [ldso/ldso/ldso.oS] Error 1
```

Though problems of this sort can be quite complex, the mechanism for attempting to solve them is pretty straightforward. In this case, if this instruction should actually be supported on the target processor, I could try a newer version of the assembler (which is included in the binutils package), or I could simply reconfigure uClibc so that it does not attempt to use this instruction.

For uClibc-related configuration issues, the easiest thing to try initially is to reconfigure uClibc. Unfortunately, uClibc configuration is run automatically for you during the buildroot process and is not repeated once a configuration file exists. To manually reconfigure uClibc, you can simply change directory to buildroot's `toolchain_build_`*ARCH*`/uClibc` subdirectory (where *ARCH* is the architecture you've selected) and run uClibc's configuration utility by executing the `make menuconfig` command. This displays a dialog like the one shown in Figure 14-15, which enables you to change the configuration settings that you previously selected.

Luckily, in my case, the option that I am looking for is displayed on the main screen, so I can simply deselect it using the spacebar and press Escape to exit the reconfiguration process (saving my changes, of course). Once you have updated the uClibc configuration file, you'll need to copy it to the directory `../../toolchain/uClibc`, giving it the name `uClibc.config`, if you are not using internationalization support; or the name `uClibc.config-locale`, if you are building with support for internationalization. You can then run `make clean` from the buildroot directory and restart the build process using the standard `make` command. Depending on the point at which you encountered problems, you may need to rerun the `make distclean` command and force buildroot to redownload and rebuild all of the packages used by the cross-compiler.

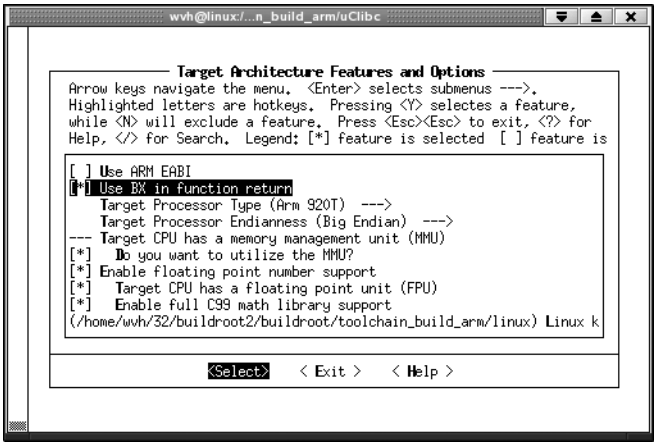

wvh@linux:/...n_build_arm/uClibc
Target Architecture Features and Options
Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> selectes a feature,
while <N> will exclude a feature. Press <Esc><Esc> to exit, <?> for
Help, </> for Search. Legend: [*] feature is selected [ ] feature is
[ ] Use ARM EABI
[*] Use BX in function return
Target Processor Type (Arm 920T) --->
Target Processor Endianness (Big Endian) --->
--- Target CPU has a memory management unit (MMU)
[*] Do you want to utilize the MMU?
[*] Enable floating point number support
[*] Target CPU has a floating point unit (FPU)
[*] Enable full C99 math library support
(/home/wvh/32/buildroot2/buildroot/toolchain_build_arm/linux) Linux k
<Select> < Exit > < Help >

Figure 14-15. *Manually reconfiguring uClibc for buildroot*

You may need to repeat the uClibc reconfiguration process multiple times if you encounter additional problems during the compilation process. For example, when building a little-endian version of the compiler for another ARM920T platform, I had to correct the endian-ness assumed by uClibc, since uClibc assumed that it was to be a big-endian compiler.

In general, the buildroot package does a nice job of preserving the configurability of the underlying packages that it uses while still providing a high-level build environment for cross-compilers, BusyBox, and embedded filesystems that use BusyBox.

Building Cross-Compilers Manually

While the crosstool and buildroot packages simplify the process of building and installing a cross-compiler, the downside is that they do not always work, especially if you are building a cross-compiler for a platform that they don't support in the first place, or if you are trying to build a cross-compiler that uses versions of the binutils, GCC, C library, and Linux kernel that these tools don't support. Though it would certainly be nice, it is impossible to anticipate problems that may arise as a result of newer software and resolve them in advance. Cutting-edge cross-compilation toolchains may require actual changes to the package source code, which you may eventually deliver back to the community as a patch—but for now, you just want to get things working.

Sometimes, you just have to build a cross-compilation toolchain manually. The fact that tools such as crosstool and buildroot exist shows that this is a well-understood and straightforward process (aside from the occasional need to fix bugs in the source code for various packages). This section outlines the steps required to manually build a cross-compilation toolchain that uses Glibc and provides information about how to integrate alternate C libraries into the toolchain build process whenever possible.

Building a cross-compiler manually is a simple process if your target architecture and processor are supported throughout all of the components of the toolchain. The source packages that you will need in order to build your own cross-compiler are the following:

- *binutils*: The tarball is available from `ftp://ftp.gnu.org/gnu/binutils`.
- *GCC*: This is available from `ftp://ftp.gnu.org/gnu/gcc`.

- *Glibc and any add-ons that you want to incorporate, or another C library:* The latest Glibc and add-ons are always available from `ftp://ftp.gnu.org/gnu/glibc`. (Glibc add-ons are explained in Chapter 12.) The sites from which you can retrieve alternate C libraries are explained in the portions of Chapter 12 that discuss those C libraries.
- *Linux kernel source code for the version of the kernel that you want to use with your cross-compiler*: Every Linux kernel source release for the past few years is available at `http://www.kernel.org`. It does not have to be exactly the same as the version of the kernel that you are using, but must at least be of the same major kernel revision (2.4, 2.6, and so on) as any kernel that you hope to build.

You will also need any available patches for any of these packages that add support for or fix bugs in that package on your target platform. When building cross-compilers for any platform other than x86 platforms, you should always check Web sites related to your architecture, such as `http://www.linux-mips.org` and `http://www.arm.linux.org.uk`. If you are using an MMU-less target, you may want to check `http://www.uclinux.org` for any patches that have not yet been integrated into the Linux kernel source that you are using. You should also check the Web site for the semiconductor that manufactured your target hardware, since it may also provide patches.

In the following description of the toolchain build process, I will use a few environment variables to simplify the example. If you are following along, make sure that you set the environment variable `TARGET`to your cross-compilation target; the environment variable `DEST` to the installation directory for your toolchain; and add the `bin` directory for your installation directory to your path (it doesn't matter if it doesn't already exist), as in the following examples for a PowerPC toolchain:

```
$ export TARGET=powerpc-linux
$ export DEST=/tools/cross-compilers/powerpc
$ export PATH=/tools/cross-compilers/powerpc/bin:$PATH
```

■ **Tip** To simplify the build and installation processes, make sure that the specified installation directory already exists and that you have write access to it.

Building a cross-compiler requires the following steps:

1. Retrieve the binutils sources; extract the contents of the tarball into your working directory; create a directory in which to build binutils; and change directory to that directory. Configure and build the binutils package with a command such as the following:

   ```
   $ ../binutils-version/configure --target=$TARGET --prefix=$DEST
   $ make
   $ make install
   ```

2. Retrieve the tarball for the version of the Linux kernel that you want to use and extract its contents in your working directory. Next, change directory into the kernel source directory and configure the kernel for your target architecture, as in the following example for the PowerPC architecture:

   ```
   $ ARCH=ppc make menuconfig
   ```

   You should visit at least the Processor submenu of the configuration utility in order to verify that you have correctly specified any process-specific kernel configuration options, since these may have some configuration impact. You should then exit and save your updated kernel configuration file for future reference.

■ **Tip** You can determine the right names for the architectures supported by the kernel by listing the contents of the `arch` subdirectory in your kernel source tree.

3. Recursively copy your configured kernel's `include/linux` and `include/asm` directories to the target location for your new toolchain, as in the following example:

```
$ mkdir $DEST/include
$ cp -rvL include/linux include/asm $DEST/include
```

4. Retrieve the tarball for the version of GCC that you want to use and extract its contents in your working directory. Next, create a directory in which to build the GCC compilers, and change directory to that directory. Configure, build, and install the first stage of the gcc C cross-compiler with a command such as the following:

```
$ ../gcc-version/configure --target=$TARGET --prefix=$DEST \
    --with-headers=$DEST/include --enable-languages="c" -Dinhibit_libc
$ make
$ make install
```

   At this point, you have a compiler that does not have Glibc, but is suitable for use in building it. This compiler could be actually used to build a kernel at this point, because there is no userspace (C library) code in the kernel.

5. Retrieve the tarball for the version of Glibc that you want to use and extract its contents in your working directory. Next, create a directory in which to build the Glibc, and change directory to that directory. Configure, build, and install Glibc using the first-stage compiler you just created with commands such as the following:

```
$ ../glibc-version/configure $TARGET --target=$TARGET --prefix=$DEST \
      --enable-add-ons  --disable-sanity-checks
$ CC=powerpc-linux-gcc make
$ CC=powerpc-linux-gcc make install
```

6. Next, change directory back to the directory that you used for building the first stage of gcc. Reconfigure, build, and install the full gcc C cross-compiler with a command such as the following:

```
$ ../gcc-version/configure --target=$TARGET --prefix=$DEST \
    --with-headers=$DEST/include --enable-languages="c"
$ make all
$ make install
```

At this point, you should have a working cross-compiler. Admittedly, you have to be lucky to find the right patches for the packages involved in building a complete cross-compilation toolchain for some platforms, but it has been done many times before, and you can do it too, with some effort. Don't forget: if you run into problems, search engines such as Google and Clusty are your new best friends.

APPENDIX A

Using GCC Compilers

This appendix provides a general discussion of how to use GCC compilers. Originally this information was in the front of the book, but I believe that most readers just want to get started learning about and using specific compilers—not necessarily wading through a generic chapter on using GCC compilers in general. Thus, I have moved this to an appendix that provides a general reference. I have extracted quick "refresher courses" on common options from this appendix and put them in the chapters that discuss each specific compiler. I hope that works for you—if not, my apologies, and please let me know.

The goals of this appendix are to introduce you to how to specify options to a GCC compiler, to introduce shared options that you can use regardless of the specific compiler in the GCC family that you are using, and to explain how to modify the behavior of the GCC compilers using environment variables and spec strings. One of the big advantages of using a family of compilers that are produced from a single base of source code is that they share a large number of options, command-line syntax, and capabilities, regardless of whether you are compiling C, Objective C, C++, Fortran, or Java code. This appendix highlights the GCC command-line options that you can use anywhere, regardless of the language that you are working in. In some cases, an option is shared by most but not all of the GCC compilers—I will still treat these as a shared option and highlight the exception(s).

When you invoke any GCC compiler to compile a source code file, the compilation process passes through up to four stages: preprocessing, compilation, assembly, and linking. The first occurs for any GCC compiler, with the exception of GCC's Java compiler. The next two occur for any input source file in any language, and the fourth is an optional stage that combines code produced by the first three stages into a single, executable file. This appendix explains how to stop the compilation process at any of these stages or specify options that control the behavior of each of these compilation stages. Other options discussed in this appendix enable you to control the names and types of output files produced when compiling your applications and also enable you to exercise greater control over the content and format of any GCC compiler's diagnostic messages.

This appendix also provides a section discussing how to modify the behavior of GCC compilers by setting environment variables or modifying entries in the specification files that tell the GCC compilers what types of files to look for during the compilation process and what to do with them. It concludes with a complete listing of all of the options to GCC compilers that are generic to all compilers, for your reference and reading pleasure.

Using Options with GCC Compilers

All GCC compilers accept both single-letter options, such as `-o`, and multiletter options, such as `-ansi`. The consequence of GCC accepting both types of options is that, unlike many GNU programs, you cannot group multiple single-letter options. For example, the multiletter option `-pg` is not the same as the two single-letter options `-p -g`. The `-pg` option creates extra code in the final binary that outputs profile information for the GNU code profiler, gprof. The combination of the `-p` and –g options,

on the other hand, generates extra code in the resulting binary that outputs profiling information for use by the prof code profiler (`-p`) and causes GCC compilers to generate debugging information using the operating system's normal format (`-g`).

Despite their sensitivity to the grouping of multiple single-letter options, GCC compilers generally enable you to mix the order of options and arguments. For example, invoking GCC's C compiler as

```
gcc -pg -fno-strength-reduce -g myprog.c -o myprog
```

has the same result as

```
gcc myprog.c -o myprog -g -fno-strength-reduce -pg
```

I say that compilers generally enable you to mix options and arguments because, in most cases, the order of options and their arguments does not matter. In some situations order does matter if you use several options of the same kind. For example, the GCC C compiler's `-I` option specifies an extra directory to search for include files. So if you specify `-I` several times, GCC searches the listed directories in the order specified.

Many options have long names starting with `-f` or with `-W`. Examples include `-fforce-mem`, `-fstrength-reduce`, `-Wformat`, and so on. Similarly, most of these long name options have both positive and negative forms. Thus, the negative form of `-fstrength-reduce` would be `-fno-strength-reduce`.

■ **Note** The GCC manual documents only the nondefault version of long name options that have both positive and negative forms. That is, if the GCC manual documents `-mfoo`, the default is `-mno-foo`.

General Information Options

Various GCC command-line options can be used to display basic or extended usage tips and compiler configuration information or control overall behavior. Basically, the options I discuss in this section do not fit neatly into any other category, so I am placing them here.

Table A-1 lists and briefly describes the options that fall into this miscellaneous category.

Table A-1. *General GCC Options*

Option	Description
`-###`	Displays the programs and arguments that would be invoked as the compiler executes with the specified command-line, but does not actually execute them. This is my favorite initial debugging option, especially in cross-platform compilation.
`-dumpmachine`	Displays the compiler's target CPU.
`-dumpspecs`	Displays GCC's default spec strings (see the section "Customizing GCC Compilers with Spec Files and Spec Strings" later in this appendix).
`-dumpversion`	Displays the compiler version number.
`--help`	Displays basic usage information.
`-pass-exit-codes`	Causes GCC to return the highest error code generated by any failed compilation phase.

Table A-1. *General GCC Options*

Option	Description
`-pipe`	Uses pipes to send information between compiler processes rather than intermediate files.
`-print-file-name=`*`lib`*	Displays the path to the library named lib, where lib is a library that is part of the GCC installation.
`-print-libgcc-file-name`	Displays the name of the compiler's companion library.
`-print-multi-directory`	Displays the root directory for all versions of libgcc.
`-print-multi-lib`	Displays the maps between command-line options and multiple library search directories.
`-print-prog-name=`*`prog`*	Displays the path to the program named prog, where prog is an application that is part of the GCC installation.
`-print-search-dirs`	Displays the directory search path.
`-save-temps`	Saves intermediate files created during compilation.
`--target-help`	Displays help for command-line options specific to the compiler's target.
`-time`	Displays the execution time of each compilation subprocess.
`-v`	Displays the programs and arguments invoked as the compiler executes.
`-V` *ver*	Invokes version number *ver* of the compiler.
`--version`	Displays the compiler version information and short license.

Spec strings are macrolike constructs that GCC uses to define the paths and default options and arguments for the various components it calls during compilation. I will discuss spec strings in greater detail later in this appendix in the section titled "Customizing GCC Compilers with Spec Files and Spec Strings." The `-dumpversion` and `-dumpmachine` options show the compiler's version number and the processor for which it outputs code, respectively. Only one of these can be used at a time—when `-dumpversion` and `-dumpmachine` are on the same line, only the first argument gets processed.

```
$ gcc -dumpmachine
```

```
x86_64-unknown-linux-gnu
```

```
$ gcc -dumpversion
```

```
4.2.0
```

```
$ gcc --version
```

```
gcc (GCC) 4.2.0 20060508 (experimental)
Copyright (C) 2006 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There
is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
```

The `--version` argument displays more verbose version information. As you can see from this example, at the time that I captured this output, I was running an experimental version of gcc 4.2.0 on a 64-bit machine. Your output will certainly differ, though the type of information that these options provide will still be the same.

I find that the `-dumpversion` and `-dumpmachine` options are best used in scripts because their output is terse and easily parsed using standard shell utilities. The output from the `-dumpmachine` option may vary between two versions of GCC running on the same system. For example, the default compiler on 64-bit SUSE Linux version 10.0 is GCC 4.0.2. When passed the `–dumpmachine` option, it emits `x86_64-suse-linux`. GCC version 4.1.0, compiled and installed as described in Chapter 11 of this book, emits `x86_64-unknown-linux-gnu` when invoked with the `-dumpmachine` option. The various options listed in Table A-1 that begin with `-print-` enable you to determine the paths to the libraries and programs that GCC compilers use at runtime. This information can be useful when you are trying to track down a problem or want to be sure that a particular library is being used or a specific directory is being searched during the compilation process. The `-###` option can also be extremely useful in identifying path-related problems in libraries or executables.

If a GCC compiler encounters an error during any compilation phase, it exits and returns an error code of 1 to the calling process. Using the `-pass-exit-codes` option instructs GCC compilers to return the highest error code generated by any compilation phase, rather than simply 1. If you want to know how long the compiler runs in each phase of the compilation process, specify the `-time` option. The `-time` option can be especially instructive when compiling GCC compilers. In fact, this option is used as a rough gauge of the overall performance of a GCC compiler by the GCC developers: the faster that GCC can compile itself, the better the developers like it. You might also find it interesting to use `-time` when compiling the Linux kernel or any other large program that really stresses a system.

The `-pipe` option uses pipes rather than temporary files to exchange data between compilation phases. Though it does save some intermediate disk space, the purpose of the `–pipe` option is speed, not disk space conservation. As interprocess communication (IPC) mechanisms, pipes are faster than files because pipes avoid the overhead of file I/O.

Controlling GCC Compiler Output

As mentioned earlier, compiling source code potentially passes through as many as four stages: preprocessing, compilation itself, assembly, and linking. The first three stages, preprocessing, compilation, and assembly, occur at the level of individual source files for all GCC compilers with the exception of gcj, which does not use a preprocessor. The end result of the first three stages is an object file. Linking combines object files into the final executable. GCC compilers evaluate filename suffixes to select the type of compilation they will perform. Table A-2 maps filename suffixes to the type of compilation that the GCC compilers perform.

Table A-2. *GCC Operations by Filename Suffix*

Suffix	Operation
.ada	ADA source code to preprocess
.adb	ADA source code to preprocess
.c	C source code to preprocess
.C	C++ source code to preprocess
.c++	C++ source code to preprocess
.cc	C++ source code to preprocess
.class	Java bytecode, no preprocessing
.cp	C++ source code to preprocess
.cpp	C++ source code to preprocess
.CPP	C++ source code to preprocess
.cxx	C++ source code to preprocess
.f	FORTRAN 77 code to preprocess
.F	FORTRAN 77 code to preprocess
.f90	Fortran 90 code to preprocess
.f95	Fortran 95 code to preprocess
.for	FORTRAN 77 code to preprocess
.FOR	FORTRAN 77 code to preprocess
.fpp	FORTRAN 77 code to preprocess
.FPP	FORTRAN 77 code to preprocess
.i	C source code that should not be preprocessed
.ii	C++ source code that should not be preprocessed
.jar	Jar format archive file of Java source code, no preprocessing
.java	Java source code, no preprocessing
.m	Objective C source code to preprocess
.M	Objective C++ source code to preprocess
.mi	Object C source code that should not be preprocessed
.mm	Objective C++ source code to preprocess
.mii	Objective C++ source code that should not be preprocessed
.h	C header file—included but not compiled or linked
.p	Pascal source code to preprocess
.pas	Pascal source code to preprocess
.r	Rational Fortran (Ratfor) source code to preprocess

Table A-2. *GCC Operations by Filename Suffix (Continued)*

Suffix	Operation
`.s`	Assembly code
`.S`	Assembly code to preprocess
`.zip`	Zip format archive file of Java source code, no preprocessing

A filename with no recognized suffix is considered an object file to be linked. GCC's failure to recognize a particular filename suffix does not mean you are limited to using the suffixes listed previously to identify source or object files. You can use the `-x` *`lang`* option to identify the language used in one or more input files. The `lang` argument tells GCC the input language to expect to encounter in an input file regardless of its name, and is specific to different GCC compilers. For example, GCC's C compiler supports values for `lang` of `c`, `objective-c`, `c++`, `c-header`, `cpp-output`, `c++-cpp-output`, `assembler`, or `assembler-with-cpp`.

■ **Tip** The file suffixes listed for each GCC compiler are listed in the file gcc/language/lang-specs.h in the GCC source code, where *language* is one of `cp`, `ada`, `java`, `objc`, `treelang`, `objcp`, or `fortran`. If you are building your own GCC compilers and use a nonstandard file extension, you can modify this file to add your own extensions. However, using the `-x` *`lang`* option is a better approach if you are writing source code that you expect to share with others.

Table A-3 lists the command-line options you can use to exercise more control over the compilation process.

Table A-3. *GCC Output Options*

Option	Description
`-c`	Stops the compilation process before the link stage
`-E`	Terminates compilation after preprocessing
`-o` *`file`*	Writes output to the file specified by *file*
`-S`	Stops the compilation process after generating assembler code
`-x` *`lang`*	Sets the input language of subsequent files to `lang`
`-x none`	Turns off the definition of a previous `-x` *`lang`* option

When you use `-c`, the output is a link-ready object file, which has an `.o` filename extension. For each input file, GCC compilers generate a corresponding output file. Likewise, if you specify `-E`, the resulting output will be the preprocessed source code, which is sent to standard output. (GCC's Java compiler, which does not perform preprocessing since the concept is irrelevant in Java, generates no output at this point.) If you want to save the preprocessed output, you should redirect it to a file, either with command-line redirection or by using the `-o` option. If you use the `-S` option, each input

file results in an output file of assembly code with an `.s` extension. The `-o` file option enables you to specify the output filename, overriding the default output filename conventions.

Compiling a single source file using a GCC compiler is simple: just invoke the appropriate compiler, specifying the name of the source file as the argument, as in the following example of using the GCC C compiler, gcc, with a source file named myprog.c:

```
$ gcc myprog.c
$ ls -l
```

```
-rwxr-xr-x    1 wvh     users       13644 Oct  5 16:17 a.out
-rw-r--r--    1 wvh     users         220 Oct  5 16:17 myprog.c
```

The result on Linux and Unix systems is an executable file in the current directory named a.out, which you execute by typing **./a.out** in the directory containing the file. The name a.out is a historical artifact dating from C's earliest days. It stands for *assembler output* because, as you might expect, the first C-based executables were the output of assemblers. On Cygwin systems, you will wind up with a file named .a.exe that you can execute by typing either **./a** or **./a.exe** in the directory containing the file.

To define the name of the output file that a GCC compiler uses, use the `-o` option, as illustrated in the following example:

```
$ gcc myprog.c -o runme
$ ls -l
```

```
-rw-r--r--    1 wvh     users         220 Oct  5 16:17 myprog.c
-rwxr-xr-x    1 wvh     users       13644 Oct  5 16:28 runme
```

As you can see, GCC creates an executable file named runme in the current directory. The usual convention when compiling a single source file to executable format is to name the executable by dropping the file extension, so that myprog.c becomes myprog. Naturally, only the simplest programs consist of only a single source code file. More typically, programming projects consist of multiple source code files. In such a situation, you need to use the `-o` option in order to name the resulting binary, unless you intend to stick with the default a.out name. Keeping the name a.out is generally a bad thing because it does not give the user any idea what the binary actually does, and the chance of colliding with some other user's a.out binary is high.

Of course, you will also want to know how to compile multiple source files using GCC compilers. Again, the magic incantation is simple. To illustrate, suppose you are working in the C language and have a source file showdate.c that uses a function that is declared in helper.h and defined in helper.c. The standard way to compile these files, ignoring optimization, debugging, and other special cases, is the following:

```
$ gcc showdate.c helper.c
```

In this example, GCC's C compiler creates the final executable in a file named a.out on Linux and Unix systems (a.exe on Cygwin systems). In the absence of command-line options instructing otherwise, GCC compilers go through the entire compilation process: preprocessing (as appropriate, based on the type of input file), compilation, assembly, and linking. To specify the name of the output file, use the `-o` option, as in the following example:

```
$ gcc showdate.c helper.c -o showdate
```

This invocation, using `-o showdate`, leaves the compiled and linked executable in the file named showdate.

Using some of the other options listed in Table A-3 can be instructive. If you want to stop compilation after preprocessing using the `-E` option, be sure to use `-o` to specify an output filename or use output redirection. For example:

```
$ gcc -E helper.c -o helper.i
$ ls -l helper.*
```

```
-rw-r--r--    1 wvh     users       210 Oct  5 12:42 helper.c
-rw-r--r--    1 wvh     users        45 Oct  5 12:29 helper.h
-rw-r--r--    1 wvh     users     40440 Oct  5 13:08 helper.i
```

The `-o helper.i` option and argument saves the output of the preprocessor in the file helper.i. Notice that the preprocessed file is some 200 times larger than the source file. Also, bear in mind that except for the link stage, GCC compilers work on a file-by-file basis—each input file results in a corresponding output file with a filename extension appropriate to the stage at which compilation is stopped. This latter point is easier to see if you use the `-S` or `-c` options, as the following example illustrates:

```
$ gcc -S showdate.c helper.c
$ ls -l
```

```
total 20
-rw-r--r--    1 wvh     users         210 Oct  5 12:42 helper.c
-rw-r--r--    1 wvh     users          45 Oct  5 12:29 helper.h
-rw-r--r--    1 wvh     users         741 Oct  5 13:18 helper.s
-rw-r--r--    1 wvh     users         208 Oct  5 12:44 showdate.c
-rw-r--r--    1 wvh     users         700 Oct  5 13:18 showdate.s
```

In this case, I used the `-S` option, which stops compilation after the assembly stage and (in this case) leaves the resulting assembly code files, helper.s and showdate.s, which are the assembled versions of the corresponding C source code files. The next example uses `-c` to stop GCC's C compiler after the compilation process itself:

```
$ gcc -c showdate.c helper.c
$ ls -l
```

```
total 20
-rw-r--r--    1 wvh     users         210 Oct  5 12:42 helper.c
-rw-r--r--    1 wvh     users          45 Oct  5 12:29 helper.h
-rw-r--r--    1 wvh     users        1104 Oct  5 13:22 helper.o
-rw-r--r--    1 wvh     users         208 Oct  5 12:44 showdate.c
-rw-r--r--    1 wvh     users        1008 Oct  5 13:22 showdate.o
```

Finally, the following example shows how to use the `-x` option to force GCC to treat input files as source code files of a specific language. First, rename showdate.c to showdate.txt and then attempt to compile and link the program as shown here:

```
$ gcc showdate.txt helper.c -o showdate
```

```
showdate.txt: file not recognized: File format not recognized
collect2: ld returned 1 exit status
```

As you might expect, GCC's C compiler does not know how to "compile" a .txt file and compilation fails. To remedy this situation, use the `-x c` option to tell GCC that the input files following the `-x` option (showdate.txt and helper.c) are C source files, regardless of their output extension:

```
$ gcc –x c showdate.txt helper.c -o showdate
$ ls -l
```

```
total 28
-rw-r--r--    1 wvh     users          210 Oct  5 12:42 helper.c
-rw-r--r--    1 wvh     users           45 Oct  5 12:29 helper.h
-rwxr-xr-x    1 wvh     users        13893 Oct  5 13:38 showdate
-rw-r--r--    1 wvh     users          208 Oct  5 12:44 showdate.txt
```

It worked! Judicious use of the `-x` option with the `-c`, `-E`, and `-S` options enables you to exercise precise control over the compilation process. Although you do not ordinarily need to do so, you can walk through a complete compilation process one step at a time if you want to examine the output of each phase of compilation. The following example uses GCC's C compiler, gcc, but you could do the same thing with any other GCC compiler, with the exception of GCC's Java compiler, gcj, which does not do preprocessing, so you would simply skip the preprocessing step.

```
$ gcc -E helper.c -o helper.pre
$ gcc -E showdate.c -o showdate.pre
$ ls -l
```

```
total 92
-rw-r--r--    1 wvh     users          210 Oct  5 12:42 helper.c
-rw-r--r--    1 wvh     users           45 Oct  5 12:29 helper.h
-rw-r--r--    1 wvh     users        40440 Oct  5 13:44 helper.pre
-rw-r--r--    1 wvh     users          208 Oct  5 12:44 showdate.c
-rw-r--r--    1 wvh     users        37152 Oct  5 13:46 showdate.pre
```

I use the `-o` option to save the output in files with `.pre` filename extensions. Next, run the preprocessed files through the assembler:

```
$ gcc -S -x cpp-output helper.pre -o helper.as
$ gcc -S -x cpp-output showdate.pre -o showdate.as
$ ls -l
```

```
total 100
-rw-r--r--    1 wvh     users          741 Oct  5 13:47 helper.as
-rw-r--r--    1 wvh     users          210 Oct  5 12:42 helper.c
-rw-r--r--    1 wvh     users           45 Oct  5 12:29 helper.h
-rw-r--r--    1 wvh     users        40440 Oct  5 13:44 helper.pre
-rw-r--r--    1 wvh     users          700 Oct  5 13:47 showdate.as
-rw-r--r--    1 wvh     users          208 Oct  5 12:44 showdate.c
-rw-r--r--    1 wvh     users        37152 Oct  5 13:46 showdate.pre
```

This time, I use the `-o` option to save the assembler output using the filename extension `.as`. I use the `-x cpp-output` option because the assembler expects preprocessor output files to have the extension `.i` (for preprocessed C source code). Now, run the assembly code through actual compilation to produce object files:

```
$ gcc -c -x assembler helper.as showdate.as
$ ls -l
```

```
total 108
-rw-r--r--    1 wvh     users            741 Oct  5 13:47 helper.as
-rw-r--r--    1 wvh     users            210 Oct  5 12:42 helper.c
-rw-r--r--    1 wvh     users             45 Oct  5 12:29 helper.h
-rw-r--r--    1 wvh     users           1104 Oct  5 13:50 helper.o
-rw-r--r--    1 wvh     users          40440 Oct  5 13:44 helper.pre
-rw-r--r--    1 wvh     users            700 Oct  5 13:47 showdate.as
-rw-r--r--    1 wvh     users            208 Oct  5 12:44 showdate.c
-rw-r--r--    1 wvh     users           1008 Oct  5 13:50 showdate.o
-rw-r--r--    1 wvh     users          37152 Oct  5 13:46 showdate.pre
```

I use the `-x assembler` option to tell GCC's C compiler that the files helper.as and showdate.as are assembly language files. Finally, link the object files to create the executable, showdate.

```
$ gcc helper.o showdate.o -o showdate
$ ls -l
```

```
total 124
-rw-r--r--    1 wvh     users            741 Oct  5 13:47 helper.ass
-rw-r--r--    1 wvh     users            210 Oct  5 12:42 helper.c
-rw-r--r--    1 wvh     users             45 Oct  5 12:29 helper.h
-rw-r--r--    1 wvh     users           1104 Oct  5 13:50 helper.o
-rw-r--r--    1 wvh     users          40440 Oct  5 13:44 helper.pre
-rwxr-xr-x    1 wvh     users          13891 Oct  5 13:51 showdate
-rw-r--r--    1 wvh     users            700 Oct  5 13:47 showdate.ass
-rw-r--r--    1 wvh     users            208 Oct  5 12:44 showdate.c
-rw-r--r--    1 wvh     users           1008 Oct  5 13:50 showdate.o
-rw-r--r--    1 wvh     users          37152 Oct  5 13:46 showdate.pre
```

It should not take too much imagination to see that a project consisting of more than a few source code files would quickly become exceedingly tedious to compile from the command line, especially after you start adding search directories, optimizations, and other GCC options. The solution to this command-line tedium is the make utility, which is not discussed in this book due to space constraints (although it is touched upon in Chapter 7).

So what was the point of this exercise? First, it illustrates that GCC compilers, in this case GCC's C compiler, performs as advertised. More importantly, the `-E` option can be remarkably useful in C, Objective C, or C++ development when you are trying to track down a problem with a macro that does not behave as you expected. A popular C programming subgenre consists of preprocessor magic, sometimes referred to as *preprocessor abuse* or, as I like to refer to it, *Stupid Preprocessor Tricks* (with apologies to David Letterman). The typical scenario is that the preprocessor does not interpret your macro as you anticipated, causing compilation failure, error or warning messages, or bizarre runtime errors. By halting compilation after preprocessing you can examine the output, determine what you did wrong, and then correct the macro definition.

The value of `-S` becomes apparent if you want to hand-tune the assembly code the compiler generates. You can also use `-S` to see what kind of assembly output the compiler creates for a given block of code, or even a single statement. Being able to examine the compiler's assembly level output is educational in its own right and can help you debug a program that has a subtle bug. Naturally, to get the maximum benefit from GCC's assembly output feature, you have to know the target system's assembly language, or, as is more often the case, have the target CPU's reference manuals close at hand.

Controlling the Preprocessor

The options discussed in this section let you control the preprocessor used by GCC compilers, with the exception of GCC's Java compiler. Skip this section if you are using the GCC gcj compiler.

As mentioned earlier in this appendix, compilation stops after preprocessing if you specify the `-E` option to a GCC compiler. As you know, the preprocessor executes against each source code file before its output is handed off to phases of the other compilation process. Preprocessor options are listed in Table A-4.

Table A-4. *Preprocessor Options*

Option	Description
`-A-QUESTION=ANSWER`	Cancels setting *`QUESTION`* to *`ANSWER`*.
`-AQUESTION=ANSWER`	Sets the value of *`QUESTION`* to *`ANSWER`*.
`-D`*`name`*	Defines the preprocessor macro name with a value of 1.
`-D`*`name=def`*	Defines the preprocessor macro name with the value specified in *def*.
`-imacros` *`file`*	Processes *file* but only includes and preprocesses its macro definitions.
`-M`	Causes the preprocessor to output rules suitable for use with the make program rather than traditional preprocessor output. See the section titled "Alphabetical GCC Option Reference" later in this appendix for detailed information.
`-nostdinc`	Tells the compiler not to search the standard system directories for include files. Only the current directory and directories specified with the `-I` option will be searched. This option is only valid for C input files.
`-nostdinc++`	Tells the compiler not to search the standard system directories for include files. Only the current directory and directories specified with the `-I` option will be searched. This option is only valid for C++ input files.
`-std=`*`std`*	Identifies the standard to which the input file should conform, which can invoke special handling for certain constructs and sets the preprocessor's expectation of valid content. This is only used when preprocessing C and C++ input files. Valid values are `c89`, `c99`, `c9x`, `c++98`, `gnu89`, `gnu99`, `gnu9x`, `gnu++98`, `iso9899:1990`, `iso9899:199409`, `iso9899:1999`, and `iso9899:199x`.
`-U`*`name`*	Undefines any preprocessor macro name.
`-undef`	Undefines all system-specific macros, leaving common and standard macros defined.
`-w`, `-W`*`option`*	Control the type of warnings issued by the preprocessor. See the section later in this appendix titled "Enabling and Disabling Warning Messages" for detailed information.
`-Xpreprocessor` *`option`*	Passes the option specified by *`option`* to the preprocessor. Any arguments to options passed using the `-Xpreprocessor` option must themselves be preceded by a separate `-Xpreprocessor` option.

The -U*name* option cancels any definition of the macro name that was defined on a GCC compiler's command line using -D or in one of the source code files. Each instance of -D and -U is evaluated in the order specified on the command line. If you use the -imacros *file* option to specify that macros defined in *file* should be included, this inclusion takes place after all -D and -U options have been evaluated.

■ **Caution** Do not put spaces between -D and -U and their arguments or the definition will not work.

Consider Listing A-1. If the preprocessor macro DEUTSCH is defined, the output message will be "Hallo, Welt!" Otherwise, the message will be "Hello, World!"

Listing A-1. *A Sample C File Showing the Use of Preprocessor Macros*

```
#include <stdio.h>

int main (void)
{
#ifdef DEUTSCH
        puts ("Hallo, Welt!");
#else
        puts ("Hello, World!");
#endif
        return 0;
}
```

In the following example, I leave DEUTSCH undefined, so the program outputs the English language greeting:

```
$ gcc hallo.c -o hallo
$ ./hallo
```

```
Hello, World!
```

Specifying -DDEUTSCH on the GCC command line, however, defines the DEUTSCH macro with a value of 1, causing the compiled binary to issue the German version:

```
$ gcc hallo.c -o hallo -DDEUTSCH
$ ./hallo
```

```
Hallo, Welt!
```

With a little command-line or Makefile magic, deftly implemented macros, and the -D and -U options, you can enable and disable program features without having to edit your code simply by enabling and disabling preprocessor macros when you recompile a program. Of course, overuse of #ifdef...#endif blocks in code can make the code unreadable, so use them sparingly.

■ **Tip** Using a construct like `#if 0...#endif` provides a convenient way to comment out huge blocks of code without using the relevant programming language comment characters. Many languages do not support nested comments, most notably C and C++, which can make it tricky to comment out blocks of code that already contain comments. The `#if 0...#endif` construct is a slick way to work around this temporarily. By the way, remember to insert a comment at the beginning of the block explaining what you are doing.

Modifying Directory Search Paths

All of the GCC compilers search directories for various libraries. Some, such as GCC's C and C++ compilers, also search for definition (include) files. The basic GCC compiler framework provides options that enable you to manipulate the list of directories to search and the order in which they are searched. The extent to which this applies to the GCC compiler that you are using depends on that particular compiler. The examples in this section use the GCC C compiler, gcc, because modifying the default system directory search paths is most common when developing C language applications. Table A-5 lists the command-line options for modifying various directory search paths.

Table A-5. *Options for Modifying Directory Search Paths*

Option	Description
`-B` *prefix*	Instructs the compiler to add `prefix` to the names used to invoke its executable subprograms (such as cpp, cc1, as, and ld), libraries, and header files.
`-iquote` *dir*	Adds `dir` to the beginning of the list of directories searched for include files requested with `#include "file"`. Directories added using `-iquote` are not searched for include files specified via `#include <file>` (GCC 4.x and greater only; C, C++, and Objective C only).
`-I` *dir*	Adds `dir` to the list of directories searched for header files.
`-I-`	Limits the type of header files searched for when `-I` *dir* is specified. (Not available in GCC 4.x; replaced by the `-iquote` option's ability to restrict searching include directories to user header files).
`-L` *dir*	Adds `dir` to the list of directories searched for library files.
`-specs=`*file*	Reads the compiler spec file *file* after reading the standard spec file, making it possible to override the default values of arguments passed to GCC component programs.

If you use `-I` *dir* to add `dir` to the include directory search list, GCC compilers insert `dir` at the beginning of the standard include search path. This enables GCC compilers to search directories containing local and custom header files before searching the standard system include directories, enabling you to override system definitions and declarations if you choose. More often, however, you use `-I` to add directories to the header file search path rather than to override already defined functions. For example, if you are compiling a program that uses header files installed in `/usr/local/include/libxml2`, you would specify this extra directory as shown in the following example (the ellipsis indicates other arguments omitted for the sake of brevity):

```
$ gcc -I/usr/local/include/libxml2 [...] resize.c
```

This command causes the GCC C compiler, gcc, to look in `/usr/local/include/libxml2` for any header files included in resize.c before it looks in the standard header file locations.

The C, C++, or Objective C languages can include two different types of header files: *system header files*, which are those included using angle brackets (for example, #include <net/inet.h>), and *user header files*, which are those included using double quotes (for example, #include "log.h"). The default behavior of the `-I` option is to search the specified directories for both user and system header files. However, you can use the `-iquote` option to modify this behavior. All directories specified using the `-iquote` option will only be searched for user header files (those included using double quotes). Any include directories specified with `-I` will still be searched for all header files.

Consider the following #include directives at the top of a sample source code file named resize.c:

```
#include "libxml2/xmlops.h"
#include <netdev/devname.h>
```

If the directories containing the include files for both of these packages are subdirectories of the include directory `/usr/local/include`, you could compile this using the following gcc invocation:

```
$ gcc -I /usr/local/include resize.c
```

This would cause the include files /usr/local/include/netdev/devname.h and /usr/local/include/libxml2/xmlops.h to be used. However, suppose that these are header files that exist in multiple places and you want to use the version of libxml2/xmlops.h located in a custom directory but ignore the version of netdev/devname.h located under that same custom directory. Assuming that this custom directory is named `working`, you could execute the following command:

```
$ gcc –iquote working -I /usr/local/include resize.c
```

Because the directory `working` is specified on the command-line using `–iquote`, the include files working/libxml2/xmlops.h and /usr/local/include/netdev/devname.h will be used. Consider what would happen if the following #include directives were used in resize.c:

```
#include <libxml2/xmlops.h>
#include <netdev/devname.h>
```

In this case, both header files are included as system header files, and gcc would therefore use the versions located under the `/usr/local/include` directory that you specified using `–I`, even if you specified the `working` directory using the `–iquote` option.

■ **Tip** Multiple `-I` *`dir`* options can be specified. They are searched in the order specified, reading left to right.

■ **Note** GCC compilers prior to version 4 used the `–I-` option to differentiate between directories that should be searched for system and user header files. All included directories specified with `-I` before the occurrence of `-I-` would only be searched for user header files (those included using double quotes). Any include directories specified with `-I` after the occurrence of `-I-` would be searched for all header files. This option is deprecated and no longer supported in GCC 4.x compilers.

The `-L` *`dir`* option does for library files what `-I` *`dir`* does for header files: it adds `dir` to the beginning of the library directory search list, so that GCC compilers first search this directory for libraries specified using the `-l` option. The `–l` option is discussed in the section "Controlling the Linker."

The `-specs=`*`file`* option will be discussed later in this appendix in the section titled "Customizing GCC Compilers Using Spec Files and Spec Strings." After learning how to use spec strings, you can store multiple spec strings in a file and apply them as a group by replacing *file* with the name of the file containing the updated spec strings.

Passing Options to the Assembler

If you have options that you want GCC compilers to ignore and pass through to the assembler, use the `-Wa,`*`opt`* option. Like its sibling option for the linker, `-Wl,`*`opt`* (discussed in the next section, "Controlling the Linker"), you can specify multiple `opt` options by separating each option with a comma.

Controlling the Linker

Linking is the last step in the compilation process and refers to merging various object files into a single executable binary. GCC compilers assume that any file that does not end in a recognized suffix is an object file or a library. Refer to the list of recognized filename suffixes listed in the section titled "Controlling GCC Compiler Output" earlier in this appendix if you need a quick refresher. The linker knows how to tell the difference between object files (`.o` files) and library files (`.so` shared libraries or `.a` archive files) by analyzing the file contents. Note that options for controlling the linker will be ignored if you use the `-E`, `-c`, or `-S` options, which terminate the compiler before the link stage begins (after preprocessing, object file generation, and assembling, respectively). Table A-6 lists the options that you can use to control the GCC linker.

Table A-6. *Link Options*

Option	Description
`-l`*`name`*	Searches the library named *name* when linking.
`-nodefaultlibs`	Specifies not to use the standard system libraries when linking.
`-nostartfiles`	Ignores the standard system startup files when linking. System libraries are used unless `-nostdlib` is also specified.
`-nostdlib`	Specifies not to use the standard system startup files or libraries when linking (equivalent to specifying `-nostartfiles -nodefaultlibs`).
`-s`	Strips all symbol table and relocation information from the completed binary.
`-shared`	Produces a shared object that can then be linked with other objects to form an executable.
`-shared-libgcc`	Uses the shared libgcc library, if available, on systems that support shared libraries.
`-static`	Forces linking against static libraries on systems that default to linking with shared libraries.
`-static-libgcc`	Uses the statically linked libgcc library, if available, on systems that support shared libraries.
`-u` *`sym`*	Behaves as if the symbol `sym` is undefined, which forces the linker to link in the library modules that define it.
`-Wl,`*`opt`*	Passes `opt` as an option to the linker.
`-Xlinker` *`opt`*	Passes `opt` as an option to the linker.

I have already mentioned the `-L` *`dir`* option for adding directories to the library search patch. The complementary *`-lname`* option enables you to specify additional library files to search for given function definitions. Each library specified with *`-lname`* refers to a file named lib*name*.a or lib*name*.so, which is searched for in the standard library search path, plus any additional directories specified by `-L` options. Most libraries are simple archive files that contain a collection of object files and are produced by the Linux/Unix ar utility. The linker processes the archive file by searching it for members that define symbols (function names) which have been referenced but not yet defined.

The differences between specifying an object filename (such as name.o) and using an *`-lname`* option are that

- Specifying `-l` embeds *name* between `lib` and the library suffix (either `.a` or `.so` if you are using a version of gcc that was built with shared library support).
- The format of library files and object files differs.
- Using `-l` searches several directories for the resulting library, based on your library search path.

For example, provided you have created the archive file libmy.a (using the ar program), the following two gcc invocations are equivalent:

```
$ gcc myprog.o libmy.o -o myprog
$ gcc myprog.o -o myprog -L. –lmy
```

Listings A-2, A-3, and A-4 show some sample code that I will use to verify the equivalence of these two gcc command lines.

Listing A-2. *A Sample C Program, swapme.c*

```
#include <stdio.h>
#include "myfile.h"

int main (void)
{
        int i = 1;
        int j = 2;

        printf ("%d %d\n", i, j);
        swap (&i, &j);
        printf ("%d %d\n", i, j);

        return 0;
}
```

Listing A-3. *A Sample Include File, myfile.h*

```
void swap(int *, int *);
```

Listing A-4. *A Sample C File, myfile.c*

```
#include "myfile.h"
void swap (int *i, int *j)
{
        int t = *i;
        *i = *j;
        *j = t;
}
```

You can demonstrate the equivalence of building and linking the code in Listings A-2, A-3, and A-4 as stand-alone object files and as an object file and a library:

1. Compile myfile.c and swapme.c to object code.

```
$ gcc -c myfile.c
$ gcc -c swapme.c
```

2. Link the resulting object files into the final binary, swapme, and then run the program to demonstrate that it works.

```
$ gcc myfile.o swapme.o -o swapme
$ ./swapme
```

```
1 2
2 1
```

3. Delete the binary.

```
$ rm swapme
```

4. Use the ar command to create an archive file named libmy.a from the myfile.o object file.

```
$ ar rcs libmy.a myfile.o
```

5. Link the object file swapme.o and the archive file as shown in the following command. This command tells the GCC C compiler to search the current directory ('.') for libraries and to load the contents of a library named libmy.a.

```
$ gcc swapme.o -o swapme -L. -lmy
```

6. Run the program again to convince yourself that it still works.

```
$ ./swapme
```

```
1 2
2 1
```

It makes a difference where in the command line you specify *-lname* because the linker searches for and processes library and object files in the order they appear on the command line. Thus, foo.o -lbaz bar.o searches library file libbaz.a after file processing foo.o but before processing bar.o. If bar.o refers to functions in libbaz.a, those functions may not be loaded.

■ **Tip** If you are specifying your own libraries and not trying to override standard libraries used by your GCC compiler, a good general rule is to add library references at the end of a GCC compiler command line to ensure that they are searched for references after all source files have been compiled or examined for external references.

If for some reason you do not want the linker to use the standard libraries, specify `-nodefaultlibs` and specify the `-L` and `-l` options to point at your replacements for functions that are traditionally defined in the standard libraries. The linker will disregard the standard libraries and use only the libraries you specify. The standard startup files will still be used, though, unless you also use `-nostartfiles`.

As noted in Table A-6, `-nostdlib` is the equivalent of specifying both `-nostartfiles` and `-nodefaultlibs`; the linker will disregard the standard startup files and only the libraries you specified with `-L` and `-l` will be passed to the linker.

■ **Note** When you specify `-nodefaultlib`, `-nostartfiles`, or `-nostdlib`, GCC still might generate internal calls to `memcmp()`, `memset()`, and `memcpy()` in System V Unix and ISO C environments, or calls to `bcopy()` and `bzero()` in BSD Unix environments. These entries are usually resolved by entries in the standard C library (libc). You should supply entry points for `memcmp()`, `memset()`, `memcpy()`, `bcopy()`, and `bzero()` when using one of these three linker options.

One of the standard libraries bypassed by `-nostdlib` and `-nodefaultlibs` is libgcc.a. The libgcc.a library contains internal subroutines that GCC compilers use to compensate for shortcomings of particular systems and to provide for special needs required by some languages. As a result, you need the functions defined in libgcc.a even when you want to avoid other standard libraries. Thus, if you specify `-nostdlib` or `-nodefaultlibs`, make sure you also specify `-lgcc` as well to avoid unresolved references to internal GCC library subroutines.

The `-static` and `-shared` options are only used on systems that support shared libraries. GCC supports static libraries on all systems that I have ever encountered. Not all systems, however, support shared libraries, often because the underlying operating system does not support dynamic loading.

On systems that provide libgcc as a shared library, you can specify `-static-libgcc` or `-shared-libgcc` to force the use of either the static or the shared version of libgcc, respectively. There are several situations in which an application should use the shared libgcc instead of the static version. The most common case occurs when compiling C++ and Java programs that may throw and catch exceptions across different shared libraries. In this case, all libraries as well as the application itself should use the shared libgcc. Accordingly, the g++ (c++) and gcj compiler drivers automatically add `-shared-libgcc` whenever you build a shared library or a main executable and would ordinarily be using the static version of libgcc by default.

On the other hand, the driver for the GCC C Compiler, gcc, does not always link against the shared libgcc, especially when creating shared libraries. The issue is one of efficiency. If gcc determines that you have a GNU linker that does not support the link option `--eh-frame-hdr`, gcc links the shared libgcc into shared libraries. If the GNU linker does support the `--eh-frame-hdr` option, gcc links with the static version of libgcc. The static version allows exceptions to propagate properly through such shared libraries, without incurring relocation costs at library load time.

In short, if a library or the primary binary will throw or catch exceptions, you should link it using the g++ (c++) compiler driver (if you are using C++) or the gcj compiler driver (if you are using Java). Otherwise, use the option `-shared-libgcc` so that the library or main program is linked with the shared libgcc.

The final option controlling the linker is the rather odd looking `-Wl,`*`opt`*. This tells GCC to pass the linker option `opt` through directly to the linker. You can specify multiple `opt` options by separating each one with a comma (`-Wl,`*`opt1,op2,opt3`*).

Enabling and Disabling Warning Messages

A warning is a diagnostic message that identifies a code construct that might potentially be an error. GCC also emits diagnostic messages when it encounters code or usage that looks questionable or ambiguous. For the sake of discussion, I divide the GCC compilers' handling of warning messages into two groups: general options that control the number and types of warnings that a compiler emits, and options that affect language features or that are language-specific. I will start with the

options that control the overall handling of warnings by GCC compilers. Table A-7 shows the most commonly used options that control warnings (see the "Alphabetical GCC Option Reference" section of this appendix for the complete mind-numbing list). Some of these options are specific to certain GCC compilers, most commonly GCC's C compiler—the explanation of the warning should make it clear whether an option applies to the GCC compiler that you are using. Specifying warning options that do not apply to your compiler is not flagged as an error by the GCC compilers.

Table A-7. *General GCC Warning Options*

Option	Description
`-fsyntax-only`	Performs a syntax check but does not compile the code.
`-pedantic`	Issues all warnings required by ISO standards and rejects GNU extensions, traditional C constructs, and C++ features used in C code.
`-pedantic-errors`	Converts warnings issued by `-pedantic` into errors that halt compilation.
`-w`	Disables all warnings, including those issued by the GNU preprocessor.
`-W`	Displays extra warning messages for certain situations.
`-Wall`	Enables all of the warnings about code constructions that most users consider questionable, dangerous, or easy to eliminate with code modifications. It was originally intended to cause the compiler to display all warnings, but there are now so many types of possible warnings that there are exceptions to this rule. For example, this option does not activate some of the more granular formatting warnings such as `–Wformat=2`, `-Wformat-nonliteral`, `Wformat-security`, and `-Wformat-y2k`, the additional warning cases specified by `–Wextra`, and many stylistic C++ warnings.
`-Wbad-function-cast`	Emits a warning if a function call is cast to an incompatible type (C only).
`-Wcast-qual`	Displays a warning when a typecast removes a type qualifier.
`-Wchar-subscripts`	Emits a warning when a `char` variable is used as a subscript.
`-Wcomment`	Emits a warning when nested comments are detected.
`-Wconversion`	Emits a warning if a negative integer constant is assigned to an unsigned type.
`-Wdisabled-optimization`	Displays a warning when a requested optimization is not performed.
`-Werror`	Converts all warnings into hard errors that halt compilation of the indicated translation unit.

Table A-7. *General GCC Warning Options (Continued)*

Option	Description
`-Werror-implicit-sfunction-declaration`	Emits an error when a function is not explicitly declared before first use.
`-Wextra`	Emits the same warnings as `-W`, but provides a more memorable option name.
`-Wfloat-equal`	Emits a warning when floating-point values are compared for equality.
`-Wformat`	Issues a warning when arguments supplied to `printf()` and friends do not match the specified format string.
`-Wformat=2`	The same as specifying `-Wformat -Wformat -nonliteral Wformat-security -Wformat-y2k`.
`-Wformat-nonliteral`	Issues a warning, if `-Wformat` is also specified, about any formatting strings that are not literal strings and therefore cannot be checked.
`-Wformat-security`	Issues a warning, if `-Wformat` is also specified, about arguments to `printf()` and friends that pose potential security problems.
`-Wformat-y2k`	Issues a warning, if `-Wformat` is also specified, if any formatting strings in calls to `printf()` and friends would display a non-Y2K-compliant (i.e., two-digit) year.
`-Wimplicit`	Combines `-Wimplicit-int` and `-Wimplicit-function-declaration`.
`-Wimplicit-int`	Emits a warning when a declaration does not specify a type.
`-Wimplicit-function-declaration`	Emits a warning when a function is not explicitly declared before first use.
`-Winline`	Issues a warning when functions declared inline are not inlined.
`-Wlarger-than-`*n*	Emits a warning when an object larger than *n* bytes is defined.
`-Wmain`	Emits a warning if `main()`'s return type or declaration is malformed.
`-Wmissing-braces`	Displays a warning when aggregate or union initializers are improperly bracketed.
`-Wmissing-declarations`	Displays a warning if a global function is defined without being declared.
`-Wnested-externs`	Issues a warning if an extern declaration occurs in a function definition.
`-Wno-deprecated-declarations`	Disables warnings about use of features that are marked as deprecated.

Table A-7. *General GCC Warning Options (Continued)*

Option	Description
`-Wno-div-by-zero`	Disables warnings issued if (integer) division by zero is detected.
`-Wno-format-y2k`	Disables warnings about `strftime()` formats that result in two-digit years.
`-Wno-format-extra-args`	Disables warnings about extra arguments to `printf()` and friends.
`-Wno-long-long`	Disables warnings about using the `long long` type.
`-Wno-multichar`	Disables warnings issued if multibyte characters are used.
`-Wpadded`	Issues a warning when a structure is padded for alignment purposes.
`-Wparentheses`	Issues a warning about ambiguous or potentially confusing use (or misuse or disuse) of parentheses.
`-Wpoint-arith`	Issues a warning when a code operation or structure depends on the size of a function type or a `void *` pointer.
`-Wredundant-decls`	Displays a warning when an object is multiply-declared in the same scope, even in contexts in which multiple declaration is permitted and valid.
`-Wreturn-type`	Issues a warning if a function's return type is not specified or if it returns a value but is declared void.
`-Wsequence-point`	Flags code that violates C sequence point rules.
`-Wshadow`	Displays a warning if a locally declared variable shadows another local or global variable, parameter, or built-in function.
`-Wsign-compare`	Issues a warning when comparisons between signed and unsigned values might produce incorrect results because of type conversions.
`-Wstrict-protoypes`	Displays a warning if a function is defined with specifying argument types (C only).
`-Wswitch`	Emits a warning about switch statements with enumerated values for unhandled cases.
`-Wsystem-headers`	Issues warnings for code in system headers as if they occurred in your own code.
`-Wtraditional`	Emits a warning about code constructs that behave differently between ISO and traditional C and about ISO C features with no parallel in traditional C (C only).
`-Wtrigraphs`	Emits warnings if any trigraphs are encountered outside of comment blocks.
`-Wundef`	Issues a warning if an undefined identifier is used in a `#if...#endif` construct.

Table A-7. *General GCC Warning Options (Continued)*

Option	Description
`-Wuninitialized`	Emits a warning when automatic variables are used without being initialized.
`-Wunknown-pragmas`	Displays a warning if a `#pragma` is used that GCC does not recognize.
`-Wunreachable-code`	Emits a warning about code that never executes.
`-Wunused`	Combines all of the listed `-Wunused` options.
`-Wunused-function`	Issues a warning about declared functions that are never defined.
`-Wunused-label`	Issues a warning about declared labels that are never used.
`-Wunused-parameter`	Issues a warning about declared function parameters that are never used.
`-Wunused-value`	Issues a warning about computed results that are never used.
`-Wunused-variable`	Issues a warning about declared variables that are never used.

The chapters of this book that discuss using each of the compilers that are part of GCC explain the options that are related to those compilers in more detail.

In general, the group of options that fall under the `-Wunused` category are particularly helpful. In optimization passes, GCC compilers do a good job of optimizing away unused objects; but if you disable optimization, the unused cruft bloats the code. More generally, unused objects and unreachable code (detected with `-Wunreachable-code`) are often signs of sloppy coding or faulty design. My own preference is to use the plain `-Wunused` option, which catches all unused objects. If you prefer otherwise, you can use any combination of the five options that begin with `-Wunused-`.

Listing A-5 is a short example of a C program with an unused variable.

Listing A-5. *A Sample C File, unused.c, with an Unused Variable*

```
int main (void)
{
        int i = 10;
        return 0;
}
```

As you can see, the program defines the `int` variable `i`, but never does anything with it. Here is the output from the gcc compiler when compiling unused.c with no options:

```
$ gcc unused.c
```

Well, perhaps I really should have written "here is the lack of output from the gcc compiler when compiling the file unused.c with no options." Even adding the `-ansi` and `-pedantic` options does not change the compiler's output. However, here is gcc's output when I add the `–Wunused` option:

```
$ gcc -Wunused unused.c -ansi -pedantic
unused.c: In function 'main':
unused.c:3: warning: unused variable 'i'
```

Each of the warning options discussed in this section results in similar output that identifies the translation unit and line number in which a problem was detected and a brief message describing the problem. I encourage you to experiment with the warning options. Given the rich set of choices, you can debug and improve the overall quality and readability of your code just by compiling with a judiciously chosen set of warning options. Many companies specify that code must compile cleanly with `-Wall` and `-Werror` in order to be considered code complete.

Adding Debugging Information

Referring to a program he used to illustrate a point, Donald Knuth once wrote "Beware of bugs in the above code; we have only proved it correct, not tried it." Bugs are an inescapable reality of coding. Accordingly, the GCC compilers support a number of options to help you debug code. If you have used GCC compilers before, you are no doubt familiar with the `-g` and `-ggdb` options, which instruct the GCC compilers to embed debugging information in executables in order to facilitate debugging. These two options hardly exhaust GCC's repertoire of debugging support. In this section I will show you not only the `-g` and `-ggdb` switches, but also other lesser-known GCC options that augment the debugging process. Or, to put it more whimsically, this section helps you debug debugging.

I will start with Table A-8, which lists and briefly describes the most commonly used debugging options supported by the GCC compilers.

Table A-8. *Debugging Options*

Option	Description
`-d[`*mod*`]`	Generates debugging dumps at compilation points specified by *mod*. See Table A-9 for a list of valid values for *mod*.
`-fdump-class-hierarchy`, `-fdump-class-hierarchy-`*option*	Dumps the class hierarchy and vtable information, subject to the control expressed by *option*, if specified.
`-fdump-ipa-`*switch*	Dumps the tree structure for various stages of interprocedural analysis to a file, subject to the value of *switch*, which is also used to determine the file extension used for the dump files. Possible values for *switch* are `all` and `cgraph`. Specifying `all` currently only dumps the same call graph information as specifying `cgraph`, but this may change in the future.
`-fdump-translation-unit`, `-fdump-translation-unit-`*option*	Dump the tree structure for an entire unit, subject to the control expressed by *option*, if specified.
`-fdump-tree-`*switch*, `-fdump-tree-`*option*	Dump the intermediate representation of the language tree structure, subject to the controls expressed by *switch* and *option*, if specified.
`-fdump-unnumbered`	Inhibits dumping line and instruction numbers in debugging dumps (see `-d[mod]`).
`-fmem-report`	Displays memory allocation statistics for each phase of the compiler. Note that this information is about the compiler itself, not the code that is being compiled.

Table A-8. *Debugging Options (Continued)*

Option	Description
`-fpretend-float`	Pretends that the target system has the same floating-point format as the host system.
`-fprofile-arcs`	Instruments code paths, creating program dumps showing how often each path is taken during program execution.
`-ftest-coverage`	Generates data for the gcov test coverage utility.
`-ftime-report`	Displays performance statistics for each compiler phase.
`-g[`*n*`]`	Generates level *n* debugging information in the system's native debugging format (*n* defaults to 2).
`-gcoff[`*n*`]`	Generates level *n* debugging information in the COFF format (Common Object File Format) (*n* defaults to 2).
`-gdwarf`	Generates debugging information in the DWARF format (Debug With Arbitrary Record Format).
`-gdwarf+`	Generates debugging information in the DWARF format, using extensions specific to GDB, the GNU debugger.
`-gdwarf-2`	Generates debugging information in the DWARF version 2 format.
`-ggdb[`*n*`]`	Generates level *n* debugging information that only GDB can fully exploit.
`-gstabs[`*n*`]`	Generates level *n* debugging information in the STABS format (*n* defaults to 2).
`-gstabs+`	Generates debugging information in the STABS format (using extensions specific to GDB).
`-gvms[`*n*`]`	Generates level *n* debugging information in the VMS format (*n* defaults to 2).
`-gxcoff[`*n*`]`	Generates level *n* debugging information in the XCOFF format (*n* defaults to 2).
`-gxcoff+`	Generates debugging information in the XCOFF format, using extensions specific to GDB.
`-p`	Generates code that dumps profiling information used by the prof program.
`-pg`	Generates code that dumps profiling information used by the gprof program.
`-Q`	Displays the name of each function compiled and how long each compiler phase takes.
`-time`	Displays the CPU time consumed by each phase of the compilation sequence.

■ **Caution** The `-a` and `-ax` options for displaying the profiling information on basic blocks in GCC prior to version 3.0 have been removed, though they still appear in some later versions of the GCC documentation.

Quite a few options, eh? Fortunately, and as is often the case with GCC, you only have to concern yourself with a small subset of the available options and capabilities at any given time or for any given project. In fact, most of the time, you can accomplish a great deal just using `-g` or `-ggdb`. Before getting started, though, I will make short work of a few options that I will discuss elsewhere: `-fprofile-arcs`, `-ftest-coverage`, `-p`, and `-pg`. Although listed as some of GCC's debugging options, they are best and most often used to profile code or to debug the compiler itself. In connection with their use in code profiling and code analysis, I discuss these options at length in Chapter 6.

The option `-d`*mod*, referred to as the *dump option* in this section, tells GCC to emit debugging dumps during compilation at the times specified by *mod*, which can be one of almost every letter in the alphabet (see Table A-9). These options are almost exclusively used for debugging the compiler itself, but you might find the debugging dumps informative or instructional if you want to learn the deep voodoo of GCC compilers. Each dump is left in a file, usually named by appending the compiler's pass number and a phrase indicating the type of dump to the source file's name (for example, myprog.c.14.bbro). Table A-9 lists the possible values for *mod*, briefly describes the corresponding compiler pass, and includes the name of the dump file, when applicable.

Table A-9. *Arguments for the Dump Option*

Argument	Description	File
`a`	Produces all dumps except those produced by `m`, `p`, `P`, `v`, `x`, and `y`	N/A
`A`	Annotates the assembler output with miscellaneous debugging information	N/A
`b`	Dumps after computing the branch probabilities pass	*file*.14.bp
`B`	Dumps after the block reordering pass	*file*.29.bbro
`c`	Dumps after the instruction combination	*file*.16.combine
`C`	Dumps after the first if-conversion	*file*.17.ce
`d`	Dumps after delayed branch scheduling	*file*.31.dbr
`D`	Dumps all macro definitions after preprocessing	N/A
`e`	Dumps after performing SSA (static single assignment) optimizations	*file*.04.ssa, *file*.07.ussa
`E`	Dumps after the second if-conversion pass	*file*.26.ce2
`f`	Dumps after life analysis	*file*.15.life
`F`	Dumps after purging ADDRESSOF codes	*file*.09.addressof
`g`	Dumps after global register allocation	*file*.21.greg
`G`	Dumps after global common subexpression elimination (GCSE)	*file*.10.gcse
`h`	Dumps after finalizing of EH handling code	*file*.02.eh

Table A-9. *Arguments for the Dump Option (Continued)*

Argument	Description	File
`I`	Dumps after the sibling call optimization pass	*file*.01.sibling
`j`	Dumps after the first jump optimization pass	*file*.03.jump
`k`	Dumps after the register to stack conversion pass	*file*.28.stack
`l`	Dumps after local register allocation	*file*.20.lreg
`L`	Dumps after the loop optimization pass	*file*.11.loop
`M`	Dumps after the machine-dependent reorganization pass	*file*.30.mach
`m`	Prints statistics on memory usage at the end of the run	Standard error
`n`	Dumps after register renumbering	*file*.25.rnreg
`N`	Dumps after the register move pass	*file*.18.regmove
`o`	Dumps after the post-reload optimization pass	*file*.22.postreload
`p`	Annotates the assembler output with a comment indicating which pattern and alternative was used	N/A
`P`	Dumps the assembler output with the equivalent RTL (register transfer level) code as a comment before each instruction and enables `-dp` annotations	N/A
`r`	Dumps after RTL generation	*file*.00.rtl
`R`	Dumps after the second scheduling pass	*file*.27.sched2
`s`	Dumps after the first common subexpression elimination (CSE) and jump optimization pass	*file*.08.cse
`S`	Dumps after the first scheduling pass	*file*.19.sched
`t`	Dumps after the second CSE pass and post-CSE jump optimization	*file*.12.cse2
`v`	Dumps a representation of the control flow graph for each requested dump file (except file.00.rtl)	*file.pass*.vcg
`w`	Dumps after the second flow analysis pass	*file*.23.flow2
`x`	Generates RTL for a function instead of compiling it (used with `-dr`)	N/A
`X`	Dumps after the SSA dead-code elimination pass	*file*.06.ssadce
`y`	Dumps debugging information during parsing	Standard error
`z`	Dumps after the peephole pass	*file*.24.peephole2

■ **Caution** The GCC info documentation lists the `-dk` option twice, once outputting the file file.28.stack and once outputting the file file.32.stack. My testing indicates that only the file.28.stack is generated.

Although the options listed in Table A-9 are most often used for debugging the compiler itself, they can also be used as an analytic aid for any program. The `-dv` option is particularly useful because its output can be used with a third-party program to generate a graphical display that corresponds to the dump option with which it was used. For example, if you compile the program myprog.c using the option `-dCv`, you will wind up with the files myprog.c.17.ce and myprog.c.17.ce.vcg, in addition to the normal compiler output.

```
$ gcc -O -dCv myprog.c -o myprog
$ ls -l myprog*
```

```
-rw-r--r--    1 wvh     users      220 Oct  5 16:17 myprog.c
-rw-r--r--    1 wvh     users     4233 Oct 16 23:04 myprog.c.17.ce
-rw-r--r--    1 wvh     users     4875 Oct 16 23:04 myprog.c.17.ce.vcg
```

The `C` argument to `-d` dumps a flow control graph after the first if-conversion pass, which means that you need to enable optimization (see Chapter 5). The `v` argument creates a second dump file that contains the same information in a format that the Visualization of Compiler Graphs (VCG) tool can read and convert into a pretty graph. VCG is available for Unix and Windows systems. More information and downloadable versions in source and binary format are available from the VCG Web site at `http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html`. VCG is developed and maintained independently of GCC.

Customizing GCC Compilers

The default behavior of the GCC compilers is usually what most people want—if that wasn't the case, the compilers would probably behave differently. GCC usually "does the right thing" and, at least in my experience, rarely violates the principle of least surprise. As you have no doubt begun to appreciate by now, if you want GCC to do something, chances are pretty good that it has a command-line option that will make it do so. Nevertheless, what should you do if you dislike GCC's default behavior, get tired of typing the same command-line option, and do not know enough to modify GCC's code? Naturally, you customize GCC using environment variables (as explained in the next section, "Customizing GCC Compilers Using Environment Variables") or specification (spec) strings, as you will learn in the section titled "Customizing GCC Compilers Using Spec Files and Spec Strings."

Customizing GCC Compilers Using Environment Variables

This section describes environment variables that affect how the GCC compilers operate. Some of these environment variables define directories or prefixes to use when searching for various kinds of files, while others control other features of the runtime environment used by the GCC compilers. In all cases, options specified on the command line always override options specified via environment values. Similarly, GCC compiler options defined in the environment always override the options built into a specific compiler, that is, those constituting each compiler's default configuration.

Some of the options in this section are specific to certain GCC compilers—these will, of course, have no effect if you are using a GCC compiler that does not reference them. If an environment variable is specific to a certain compiler or input language, the discussion of that environment variable identifies that fact.

The first two environment variables, `COMPILER_PATH` and `GCC_EXEC_PREFIX`, control modification of GCC's search path. `COMPILER_PATH` contains a colon-delimited list of directories, comparable to the familiar Unix `PATH` environment variable. GCC uses the directories specified in `COMPILER_PATH`, if any, to find subprograms, such as cc1, collect2, and cpp0, in its default search path or subprograms with the prefix specified by `GCC_EXEC_PREFIX`.

GCC_EXEC_PREFIX, if set, specifies the prefix to use in the names of the subprograms executed by the compiler. This variable does not necessarily specify a directory name because GCC does not add a slash (/) to the front of the prefix when it combines GCC_EXEC_PREFIX with the name of a subprogram. So, if GCC_EXEC_PREFIX is /usr/lib/gcc-lib/i586-suse-linux/4.1/, GCC attempts to invoke the subprogram cc1 as /usr/lib/gcc-lib/i586-suse-linux/4.1/cc1. If GCC_EXEC_PREFIX is custom-, GCC compilers attempt to invoke the subprogram cc1 as custom-cc1.

■ **Note** The GCC_EXEC_PREFIX environment variable does not seem to work under Cygwin. I am unclear if this is a function of Cygwin's emulation environment or a problem in a specific version of GCC.

If GCC_EXEC_PREFIX is unset, GCC uses a heuristic to derive the proper prefix based on the path by which you invoked GCC. The default value is PREFIX/lib/gcc-lib/, where PREFIX is the value of --prefix you specified when you configured the GCC script. If you specify -B on the command line, as noted earlier, the command-line specification overrides GCC_EXEC_PREFIX.

■ **Note** GCC_EXEC_PREFIX also enables GCC to find object files, such as crt0.o and crtbegin.o, used during the link phase.

C_INCLUDE_PATH, used by the C preprocessor, specifies a list of directories that will be searched when preprocessing a C file. This option is equivalent to the -isystem option, except that paths specified in C_INCLUDE_PATH are searched after directories specified with -isystem.

Another environment variable, CPATH, used by the C preprocessor, specifies a list of directories to search for included files when preprocessing C programs. This option is equivalent to the -I option, except that directories specified in CPATH are searched after directories specified using -I. CPATH is always used if defined, and used before C_INCLUDE_PATH, CPLUS_INCLUDE_PATH, and OBJC_INCLUDE_PATH if they are defined.

CPLUS_INCLUDE_PATH functions comparably to C_INCLUDE_PATH, but applies to C++ compilations rather than C compilations. CPLUS_INCLUDE_PATH specifies a list of directories that will be searched during preprocessing of a C++ file. This option is equivalent to the -isystem option, except that paths specified in CPLUS_INCLUDE_PATH are searched after directories specified with -isystem.

Finally, OBJC_INCLUDE_PATH contains a list of directories to search when preprocessing Objective C source files. This option is equivalent to the -isystem option, except that paths specified in OBJC_INCLUDE_PATH are searched after directories specified with -isystem.

As discussed later in this appendix in the section titled "Alphabetical GCC Option Reference," GCC's -M options can be used to tell GCC's C compiler to generate dependency output for all nonsystem header files (system header files are ignored) for the make utility. The environment variable DEPENDENCIES_OUTPUT customizes this behavior if set. If DEPENDENCIES_OUTPUT is a filename, the generated make rules are written to that file, and GCC guesses the target name based on the name of the source file. If DEPENDENCIES_OUTPUT is specified as *FILE TARGET*, the rules are written to the file *FILE* using *TARGET* as the target name. Using DEPENDENCIES_OUTPUT is equivalent to combining the options -MM and -MF, and optionally -MT, except that using DEPENDENCIES_OUTPUT enables compilation to proceed while still saving the text-format make rules in a file.

LIBRARY_PATH is a colon-delimited list in which GCC searches for special linker files. LIBRARY_PATH is searched after GCC_EXEC_PREFIX. If you use GCC for linking, that is, if you do not invoke the linker directly, GCC uses the directories specified in LIBRARY_PATH, if any, to find libraries specified with -L and -l (in that order).

GCC tries to play nice with language and locale information, so it is sensitive to the presence of the LANG, LC_ALL, LC_CTYPE, and LC_MESSAGES environment variables. GCC uses these variables to select the character set to use when the C and C++ compilers parse character literals, string literals, and comments. If configured for multibyte characters, GCC understands JIS, SJIS, and EUCJP characters if LANG has the value C-JIS, C-SJIS, or C-EUCJP, respectively. Otherwise, GCC uses the locale functions mblen() and mbtowc() to process multibyte characters.

Whereas LANG determines how GCC compilers process character sets, LC_ALL, LC_CTYPE, and LC_MESSAGES control how GCC uses locale information to interpret and process other elements of locale customization. GCC uses LC_CTYPE, for example, to determine the character boundaries in a string—some multibyte character encodings contain quote and escape characters that are ordinarily interpreted to denote an escape character or the end of a string. LC_MESSAGES informs GCC of the language to use when emitting diagnostic messages.

If LC_CTYPE or LC_MESSAGES are not set, they default to the value of the LANG environment variable. If set, the LC_ALL environment variable overrides the values of the LC_CTYPE and LC_MESSAGES environment variables. If none of these variables are set, GCC defaults to the traditional behavior of a C compiler in a U.S. English setting.

Finally, TMPDIR defines the directory to use for temporary files, which is normally /tmp on Unix systems. GCC uses temporary files to hold the output of one compilation stage, which will be used as input to the next stage. For example, the output of the preprocessor is the input to the compiler proper. One way to speed up any GCC compiler is to define TMPDIR to point to a RAM disk or a Linux tmpfs filesystem, which bypasses the overhead and limitations of physical disk reads and writes during compilation. If TMPDIR is not set, GCC uses the default temporary directory, which varies according to system.

Customizing GCC Compilers with Spec Files and Spec Strings

One of the points I have tried to emphasize throughout this book is that all GCC compilers are actually a sequence of programs that are executed in order. The idea of writing smaller programs that do one thing and do it well, and then chaining them together, is the classic Unix programming paradigm. For example, gcc is a driver program that does its job by invoking a sequence of other programs to do the heavy lifting of preprocessing, compiling, assembling, and linking. In general, the command-line parameters and filename arguments that you pass to any GCC compiler help it decide which helper and back-end programs to invoke and how to invoke them. Spec strings control the invocation of the helper programs.

A *spec string* is a list of options passed to a program. Spec strings can contain variable text substituted into a command line, or text that might or might not be inserted into a command line based on a conditional statement (in the spec string). Using the flexible constructs provided by spec strings, it is possible to generate quite complex command lines. A *spec file* is a plain text file that contains spec strings. Spec files contain multiple directives separated by blank lines. The type of directive is determined by the first nonwhitespace character on the line. In most cases there is one spec string for each program that a GCC compiler can invoke, but a few programs have multiple spec strings to control their behavior.

■ **Note** Prior to version 4.0 of GCC, the spec strings used by default by all of the GCC compilers were generated and stored in a single text file. It was common practice to query and potentially modify aspects of the behavior of various GCC compilers by examining and modifying this file. This file was typically installed as *prefix*/lib/gcc/*system-type*/*gcc-version*/specs. For example, on a 64-bit AMD system, the specs file for a set of GCC 3.4.4 compilers installed into `/usr/local/gcc3.4.4` would have been the file /usr/local/gcc-3.4.4/lib/gcc/x86_64-unknown-linux/gnu/3.4.4/specs. With GCC 4.0 and greater, the spec strings used by the GCC compilers are no longer stored in a single text file. If you want to modify spec strings, you must first use the `-dumpspecs` option to dump the internal set of spec strings used by your compiler, redirecting the output of this command into a file. You can then modify this file to your heart's content and then load it using the `-specs=`*`file`* option.

To override the spec strings built into the GCC compilers (and after generating the specs file, if necessary), use the `-specs=`*`file`* option to specify an alternative or additional spec file named *file*. The GCC compiler driver reads the specified file after reading the defaults from the standard specs file. If you specify multiple spec files, the compiler driver processes them in order from left to right. Table A-10 lists the possible spec file directives.

Table A-10. *Spec File Directives*

Directive	Description
`%command`	Executes the spec file command specified by *command* (see Table A-11)
`*spec_name`	Creates, deletes, or overrides the spec string specified by *spec_name*
`suffix`	Creates a new suffix rule specified by *suffix*

The next few sections explain these different types of spec file directives.

Spec File Commands

Before you start to panic at the prospect of facing yet another command language to learn, fear not: GCC's spec file command language has only three very simple and easy-to-remember commands: `include`, `include_noerr`, and `rename`. Table A-11 describes what these spec file commands do.

Table A-11. *Spec File Commands*

Command	Description
`%include` *`file`*	Searches for and inserts the contents of *file* at the current point in the spec file
`%include_noerr` *`file`*	Works like `%include`, without generating an error message if *file* is not found
`%rename` *`old new`*	Renames the spec string `old` to `new`

Creating, Deleting, or Redefining Spec Strings

The `*spec_name` directive creates, deletes, or redefines the spec string `spec_name`, whose name is separated from the new definition by a colon. The spec string definition or redefinition continues to the next directive or blank line. To delete a spec string, add a blank line or another directive immediately following `*spec_name`. To append text to an existing spec, use + as the first character of the text defining the spec.

Creating, Deleting, or Modifying Suffix Rules

Spec file suffix rules create, delete, or modify a spec pair, which is comparable to a suffix rule for the make utility. Suffix rules exist primarily to simplify extending the GCC compiler driver program to handle new back-end compilers and new file types. As with spec string definitions following the * directive, the suffix in the suffix rule must be separated from the actual rule directives by a colon, with the spec string for *suffix* being defined by the text following *suffix* up to the next directive or a blank line. When the compiler encounters an input file with the named suffix, it uses the spec string to determine how to compile that file. For example, the following suffix directive creates a spec string that says any input file ending with the characters `.d` should be passed to the program dcc, with an argument of `-j` and the results of the substitution for `%i` (I discuss substitution in the section titled "Modifying and Processing Spec Strings").

```
.d:
dcc -j %i
```

The spec string `@language` following a suffix definition tells GCC that the given suffix definition is really another valid suffix for a predefined language. For example, the following suffix rule tells the GCC compilers that files whose names end in `.d` should be treated exactly as files with the `.c` suffix traditionally used by GCC's C compiler and friends:

```
.d:
@c
```

The spec string *#name* following a suffix specification instructs the GCC compiler to emit an error message that says "name compiler not installed on this system." The actual text of the error message appears to differ depending on the version of GCC you are using, the compiler that you are using, and the platform (operating system) on which you are working. For example, consider the following suffix spec string:

```
.d:
#dcc
```

In this case, if the gcc binary encounters a file whose name ends in `.d`, the GCC compiler driver will emit the error message "dcc compiler not installed on this system."

■ **Tip** If you choose to work through the examples in this section, you might find it easier to see what is happening if you invoke the compiler with `-###`. This option makes it easier to distinguish the output.

Now that you know how to create and modify spec strings, take a look at Table A-12, which lists GCC's built-in spec strings. You can use this table to learn how GCC's designers have configured GCC and, more importantly, to identify spec strings you should **not** idly redefine unless you want to break the GCC compilers in really interesting ways.

Table A-12. *Built-in Spec Strings*

Spec String	Description
`asm`	Specifies the options passed to the assembler
`asm_final`	Specifies the options passed to the assembler postprocessor
`cc1`	Specifies the options passed to the C compiler
`cc1plus`	Specifies the options passed to the C++ compiler
`cpp`	Specifies the options passed to the C preprocessor
`endfile`	Specifies the object files to link at the end of the link phase
`lib`	Specifies the libraries to pass to the linker
`libgcc`	Specifies the GCC support library (libgcc) to pass to the linker
`link`	Specifies the options passed to the linker
`linker`	Specifies the name of the linker
`predefines`	Specifies the `#defines` passed to the C preprocessor
`signed_char`	Specifies the `#defines` passed to the C preprocessor indicating if a `char` is signed by default
`startfile`	Specifies the object files to link at the beginning of the link phase

Modifying and Processing Spec Strings

In addition to extending the GCC compilers to support new suffixes, another common reason for modifying spec file strings is to change how the compiler handles files with new suffixes or to ensure that GCC uses a specific program, such as a certain cross-compiler or part of a cross-compilation chain, to process files with existing suffixes. As such, spec files support a fairly rich set of built-in strings that can be used when modifying or working with spec strings. Table A-13 shows many of GCC's predefined substitution specs.

Table A-13. *Predefined Substitution Specs*

Spec String	Description
`%%`	Substitutes a `%` into the program name or argument.
`%b`	Substitutes the currently processing input file's basename (as the `basename` shell command might generate) without the suffix.
`%B`	Substitutes the currently processing input file's basename, like `%b`, but includes the file suffix.
`%d`	Denotes the argument following `%d` as a temporary filename, which results in the file's automatic deletion upon gcc's normal termination.
`%estr`	Designates `str` as a newline-terminated error message to display.

Table A-13. *Predefined Substitution Specs*

Spec String	Description
`%gsuffix`	Substitutes a filename with a suffix matching `suffix` (chosen once per compilation), marking the argument for automatic deletion (as with `%d`).
`%i`	Substitutes the name of the currently processing input file.
`%jsuffix`	Substitutes the name of the host's null device (such as `/dev/null` on Unix systems), if one exists, if it is writable, and if `-save-temps` has not been specified. On systems that do not have a null device or some other type of bit bucket, `%jsuffix` substitutes the name of a temporary file, which is treated as if specified with the `%usuffix` spec. Such a temporary file should not be used by other spec strings because it is intended as a way to get rid of temporary or intermediate data automatically, not as a means for two compiler processes to communicate.
`%o`	Substitutes the names of all the output files, delimiting each name with spaces.
`%O`	Substitutes the suffix for object files.
`%(name)`	Inserts the contents of spec string `name`.
`%usuffix`	Substitutes a filename with a suffix matching `suffix`, like `%gsuffix`, but generates a new temporary filename even if `%usuffix` has already been specified.
`%v1`	Substitutes GCC's major version number.
`%v2`	Substitutes GCC's minor version number.
`%v3`	Substitutes GCC's patch-level number.
`%w`	Defines the argument following `%w` as the current compilation's output file, which is later used by the `%o` spec (see the entry for `%o`).

Here is a small example of a spec string that changes the linker definition from, say, `ld`, to `my_new_ld`:

```
%rename linker old_linker

*linker:
my_new_%(old_linker)
```

This example renames the spec string named `linker` to `old_linker`. The blank line ends the `%rename` command. The next line, `*linker`, redefines the `linker` spec string, replacing it with `my_new_%(old_linker)`. The syntax `%(old_linker)`, known as a *substitution*, appends the previous definition of `linker` to the new definition, much as one would use the Bourne shell command `PATH=/usr/local/gcc/bin:$PATH` to insert `/usr/local/gcc/bin` at the beginning of the existing directory search path.

Table A-14 lists the available spec processing instructions.

Good examples of files that use and modify spec files can be found in the test suites that accompany the various GCC compilers and in many of the wrapper compilation scripts that accompany alternate C libraries such as uClibc or Newlib.

Table A-14. *Spec Processing Instructions*

Spec String	Description
%1	Processes the cc1 spec, which selects the options to pass to the C compiler (cc1).
%2	Processes the cc1plus spec, which builds the option list to pass to the C++ compiler (cc1plus).
%a	Processes the asm spec, which selects the switches passed to the assembler.
%c	Processes the signed_char spec, which enables cpp (the C preprocessor) to decide if a char is signed or unsigned.
%C	Processes the cpp spec, which builds the argument list to pass to cpp.
%E	Processes the endfile spec, which determines the final libraries to pass to the linker.
%G	Processes the libgcc spec, which determines the correct GCC support library against which to link.
%l	Processes the link spec, which constructs the command line that invokes the linker.
%L	Processes the lib spec, which selects the library names to pass to the linker.
%S	Processes the startfile spec, which determines the startup files (such as the C runtime library) that must be passed to the linker first. For C programs, this is the file crt0.o.

Alphabetical GCC Option Reference

As you would expect from the world's most popular (and freely available) compiler, the compilers in the GNU Compiler Collection have a tremendous number of potential command-line options. This is partially due to the flexibility that arises from having thousands of users and contributors to GCC, each of whom wants the compiler to behave in a specific way, but even more so due to the tremendous number of platforms and architectures on which GCC is used.

GCC's online help in info format is a great source of reference and usage information for GCC's command-line options. However, option information in GCC info is organized into logical groups of options, rather than providing a simple, alphabetical list that you can quickly scan to obtain information about using a specific option. It is also sometimes somewhat out of date—we all know that the documentation is typically the last thing to be updated.

This section provides a single monolithic list of all of the machine-independent GCC command-line options, organized alphabetically for easy reference. GCC also provides hundreds of machine-specific options that you will rarely need to use unless you are using a specific platform and need to take advantage of some of its unique characteristics. Because machine-specific options are both the largest set of GCC options and the set that you will be using least frequently if you tend to work on a more standard hardware platform or operating system, I have grouped these options together in Appendix B.

Only those well-versed in character and string comparisons know offhand whether *A* is less than *a*, and so on, and it is a pain to type **man ascii** each time you need to remember how to sort a specific letter. Since not everyone may be one with the ASCII chart (and may not even be using a Linux, *BSD, or Un*x machine, for that matter), the options described in this section are listed more or less alphabetically, with single-letter uppercase options preceding single-letter lowercase options involving the same letter of the alphabet. Options whose names begin with symbols are listed before

the alphabetic options, followed by options beginning with numerals, and concluding with alphabetic options. The number of dashes preceding any given argument is shown, but is ignored for sorting purposes.

■ **Note** Language-specific options in this section identify the appropriate language whenever possible. This is intended as much for completeness' sake as to help you identify options that are irrelevant for the language that you are compiling.

`-###`: This output option causes the GCC commands relevant to your command line to be displayed in quoted form but not executed. This option is typically used to identify mode-specific commands that you can subsequently incorporate into shell scripts, or to verify exactly what GCC is attempting to execute in response to specific command-line options.

`-A-`: This option cancels all predefined assertions and all assertions that precede it on the command line, and undefines all predefined macros and all macros that precede it on the command line.

`-A-QUESTION=ANSWER`: This option cancels setting the value of `QUESTION` to `ANSWER`. This option is typically used to cancel assertions that had previously been made for use by the preprocessor.

`-AQUESTION=ANSWER`: This option sets the value of `QUESTION` to `ANSWER`. This option is typically used to make assertions for use by the preprocessor. For example, `-A system=gnu` would tell the preprocessor that the value of the system predicate should be asserted as `gnu`.

`-ansi`: This option, for C and C++ programs, enforces compliance with ANSI C (ISO C89) or standard C++, disabling features such as the `asm` and `typeof` keywords; predefined platform-specific macros such as `unix` or `vax`; the use of C++ `//` comments in C programs; and so on. When using this option, non-ISO programs can still be successfully compiled—to actively reject programs that attempt to use non-ANSI features, you must also specify the `-pedantic` option.

`-aux-info filename`: This option, for C programs, causes prototyped declarations for all referenced functions to be dumped to the specified output file filename, including those defined in header files. This option is silently ignored in any language other than C. The output file contains comments that identify the source file and line number for each declared function, and letters indicating whether the declaration was implicit (I), prototyped (N), or unprototyped (O), and whether the function was declared (C) or defined (F) there. Function definitions are followed by a list of arguments and their declarations.

`-B`*`prefix`*: This option, when using any GCC compiler as a cross-compiler, enables you to specify a prefix that should be used to try to find the executables, libraries, and include and data files for the compiler. For each of the subprograms (cpp, cc1, as, and ld) run by the compiler, using this option causes the compiler driver to try to use *`prefix`* to locate each subprogram, both with and without any values specified with the `-b` (`machine`) and `-V` (`version`) options. If binaries with the specified prefix are not found in the directories listed in your `PATH` environment variable, the GCC compiler's driver also looks in the directories `/usr/lib/gcc` and `/usr/local/lib/gcc-lib` for both binaries and subdirectories with relevant names. Using this option also causes the GCC compiler's driver to attempt to locate and use include files and libraries with the specified prefix. This command-line option is equivalent to setting the `GCC_EXEC_PREFIX` environment variable before compilation.

`-b machine`: This option enables you, when using gcc as a cross-compiler, to identify the target machine, and therefore the associated compiler. `machine` should be the same value specified when you executed the configure script to build the cross-compiler.

`-C`: This option causes the preprocessor to retain comments from the input files, with the exception of comments within preprocessor directives, which are deleted along with the directive. This option can be useful if you want to examine the output of the preprocessor (preserved by the `-save-temps` option) and need a convenient way for delimiting regions of code with comments so that you can see exactly what the preprocessor is doing with your input code.

`-CC`: This option does not discard comments when processing the input file, including those contained in macros that are inserted as a result of macro expansion.

`-c`: This output option causes the GCC compiler driver to compile or assemble the source files without linking them, producing separate object files. This option is typically used to minimize recompilation when compiling and debugging multimodule applications.

`-combine`: This option tells the GCC compiler driver to pass all input source code files to the compiler at once, and is only currently supported by the gcc C compiler. If used with the `–save-temps` option, one preprocessed file will be produced for each input file, but only one final assembler output and object code file will be produced.

`-DMACRO[=`*`DEFINITION`*`]`: This preprocessor option sets the value of *`MACRO`* to 1 if no *`DEFINITION`* is specified, or to *`DEFINITION`* if specified.

`-d`*`letters`*: This internal compiler debugging option (rather than an application debugging option) causes the GCC compiler driver to generate debugging output files during compilation at times specified by *letters*. The names of the debugging files are created by appending a pass number (pass) to a word identifying the phase of compilation to the name of the source file (file), separated by a period, and then adding an extension that reflects the phase of compilation at which the file was generated. Values for letters and the names of the output files are as follows:

- `A`: Annotates the assembler output with miscellaneous debugging information
- `a`: Produces the rtl, flow2, addressof, stack, postreload, greg, lreg, life, cfg, and jump debugging output files
- `B`: Dumps after block reordering (file.pass.bbro)
- `b`: Dumps after computing branch probabilities (file.pass.bp)
- `C`: Dumps after the first if-conversion (file.pass.ce)
- `c`: Dumps after instruction combination (file.pass.combine)
- `D`: Dumps all macro definitions at the end of preprocessing
- `d`: Dumps after delayed branch scheduling (file.pass.dbr)
- `E`: Dumps after the second if-conversion (file.pass.ce2)
- `e`: Dumps after static single assignment (SSA) optimizations (file.pass.ssa and file.pass.ussa)
- `F`: Dumps after purging `ADDRESSOF` codes (file.pass.addressof)
- `f`: Dumps after life analysis (file.pass.life)
- `G`: Dumps after global common subexpression elimination (GCSE) (file.pass.gcse)
- `g`: Dumps after global register allocation (file.pass.greg)
- `h`: Dumps after finalization of exception handling (EH) code (file.pass.eh)
- `i`: Dumps after sibling call optimizations (file.pass.sibling)
- `j`: Dumps after the first jump optimization (file.pass.jump)
- `k`: Dumps after conversion from registers to stack (file.pass.stack)
- `L`: Dumps after loop optimization (file.pass.loop)

- `l`: Dumps after local register allocation (file.pass.lreg)
- `M`: Dumps after performing the machine-dependent reorganization (file.pass.mach)
- `m`: Prints statistics on memory usage at the end of the run
- `N`: Dumps after the register move pass (file.pass.regmove)
- `n`: Dumps after register renumbering (file.pass.rnreg)
- `o`: Dumps after post-reload optimizations (file.pass.postreload)
- `P`: Dumps the register transfer language (RTL) in the assembler output as a comment before each instruction
- `p`: Annotates assembler output with a comment identifying the pattern, alternative, and length of each instruction
- `R`: Dumps after the second scheduling pass (file.pass.sched2)
- `r`: Dumps after RTL generation (file.pass.rtl)
- `S`: Dumps after the first scheduling pass (file.pass.sched)
- `s`: Dumps after first common subexpression elimination (CSE) and associated jump optimization pass (file.pass.cse)
- `t`: Dumps after the second CSE and associated jump optimization pass (file.pass.cse2)
- `v`: Dumps a representation of the control flow graph for each dump file. This representation is suitable for viewing with VCG (file.pass.vcg)
- `w`: Dumps after the second flow pass (file.23.flow2)
- `X`: Dumps after SSA dead-code elimination pass (file.pass.ssadce)
- `x`: Only generates RTL for each function instead of compiling it
- `y`: Dumps debugging information during parsing
- `z`: Dumps after the peephole pass (file.pass.peephole2)

`-dumpmachine`: This debugging option displays the compiler's target machine and then exits.

`-dumpspecs`: This debugging option displays the compiler's built-in specs and then exits. These are specifications stored in a file named specs, located in the `lib/gcc-lib` directory of the directory hierarchy in which you installed GCC. GCC's specifications tell GCC where to find various mandatory files and libraries, which tools to use at each phase of compilation, and how to invoke them.

`-dumpversion`: This debugging option displays the compiler's version and then exits.

`-E`: This output option causes the GCC framework to define the macros `__GNUC__`, `__GNUC_MINOR__`, and `__GNUC_PATCHLEVEL__`, and to stop after the preprocessing stage without running the compiler itself. Files that do not require preprocessing are ignored.

`-fabi-version=`*n*: This option specifies the version of the C++ ABI (application binary interface) that the g++ compiler should use. Version 2 is the default and is the version of the ABI introduced with g++ version 3.4. Version 1 is the version of the ABI introduced with g++ version 3.2. Version 0 is the version of the ABI that conforms most closely to the actual C++ ABI specification.

`-falign-functions` | `-falign-functions=`*n*: These optimization options cause GCC to align the start of functions to a machine-specific value (when *n* is not specified) or the next power-of-two boundary greater than *n*, skipping up to *n* bytes. Specifying `-falign-functions=1` is equivalent to specifying `-fno-align-functions`, meaning that functions will not be aligned.

`-falign-jumps` | `-falign-jumps=`*n*: These optimization options cause GCC to align branch targets to a machine-specific value (when *n* is not specified) or to the next power-of-two boundary, skipping up to *n* bytes. Specifying `-falign-jump=1` is equivalent to specifying `-fno-align-jumps`, meaning that jumps will not be aligned.

`-falign-labels` | `-falign-labels=`*n*: These optimization options cause GCC to align all branch targets to a machine-specific value (when *n* is not specified) or to the next power-of-two boundary, skipping up to *n* bytes. Specifying this option causes GCC to insert dummy options in output code to cause the requested alignment. Specifying `-falign-labels=1` is equivalent to specifying `-fno-align-labels`, meaning that labels will not be aligned. If `-falign-loops` or `-falign-jumps` are given and their machine-specific or specified values are greater than the value requested by this option, their values are used instead.

`-falign-loops` | `-falign-loops=`*n*: These optimization options cause GCC to align loop targets to a machine-specific value (when *n* is not specified) or to the next power-of-two boundary, skipping up to *n* bytes. Specifying `-falign-loops=1` is equivalent to specifying `-fno-align-loops`, meaning that loops will not be aligned.

`-fallow-single-precision`: This option, when compiling C applications, specifies not to promote single precision math operations to double precision (the K&R C default), even if `-traditional` is specified. Using this option may provide performance optimizations on certain architectures. This option has no effect when compiling with ISO or GNU C conventions.

`-falt-external-templates`: This deprecated option for C++ compilation generates template instances based on where they are first instantiated. This option is similar to `-fexternal-templates`.

`-fargument-alias` | `-fargument-noalias` | `-fargument-noalias-global`: These code generation options specify the possible relationships among parameters and between parameters and global data. These options rarely need to be used—in most cases, the GCC framework uses the options appropriate to the language that is being compiled. The `-fargument-alias` option specifies that arguments (parameters) can alias each other and global storage. The `-fargument-noalias` option specifies that arguments do not alias each other, but can alias global storage. The `-fargument-noalias-global` option specifies that arguments do not alias each other or global storage.

`-fasynchronous-unwind-tables`: This code generation option causes GCC to generate a loop unwind table in DWARF2 format (if DWARF2 is supported by the target machine) that can be used in stack unwinding by asynchronous external events such as a debugger or a garbage collector.

`-fbounds-check`: This optimization option, in GCC's Java and FORTRAN 77 front ends, automatically generates additional code that checks whether all array indices are within the declared size of the appropriate array. This option is on by default in Java, and false by default for FORTRAN 77. This option is ignored when compiling code in languages other than Java and FORTRAN 77.

`-fbranch-probabilities`: This optimization option uses the file.da files produced by a run of GCC with the `-fprofile-arcs` option to further optimize code based on the number of times each branch is taken. The information in the .da files is closely linked to the GCC options used during a specific run of GCC, so you must use the same source code and the same optimization options for both compilations.

`-fbranch-target-load-optimize`: This optimization option tells the GCC compilers to perform branch target register load optimization before prologue/epilogue threading. This hoists loads out of loops and enables interblock scheduling.

`-fbranch-target-load-optimize2`: This optimization option tells the GCC compilers to perform branch target register load optimization after prologue/epilogue threading.

`-fbtr-bb-exclusive`: This optimization option tells GCC not to reuse branch target registers within any basic block when doing branch target register load optimization.

`-fcall-saved-reg`: This code generation option tells GCC to treat the register named reg as an allocable register whose contents are saved by functions and will therefore persist across function calls. Functions compiled with this option save and restore the register reg if they use it. This option should not be used with registers such as the frame pointer or the stack pointer that are used internally by the compiler, or in which function values are returned. Registers are machine-specific—those valid for each specific GCC output target are defined in the `REGISTER_NAMES` macro in the machine description macro file.

`-fcall-used-reg`: This code generation option tells GCC to treat the register named reg as an allocable register whose contents may be overwritten by function calls. Functions compiled with this option will not save and restore the register reg. This option should not be used with registers such as the frame pointer or the stack pointer that are used internally by the compiler. Registers are machine-specific—those valid for each specific GCC output target are defined in the `REGISTER_NAMES` macro in the machine description macro files.

`-fcaller-saves`: This optimization option tells GCC to automatically add extra instructions that save and restore the contents of each register across function calls. This enables compiled code to make global use of registers that may also be used as scratch pads within various functions. This option is active by default for systems that have no call-preserved registers. This option is automatically enabled when using optimization level 2 and higher.

`-fcheck-new`: This option, when compiling C++ programs, causes GCC to check that the pointer returned by the operator `new` is nonnull before attempting to use the storage that the pointer refers to. This should be unnecessary, since `new` should never return a null pointer. If you declare your operator `new` as `throw()`, G++ will automatically check the return value.

`-fcond-mismatch`: This option, when compiling C programs, allows the successful compilation of conditional expressions with mismatched types in the second and third arguments. The value of such expressions is `void`. This option is not supported for C++.

`-fconserve-space`: This option, when compiling C++ programs, causes GCC to put uninitialized or runtime-initialized global variables into the common segment, as is done when compiling C programs. This saves space in the executable but obscures duplicate definitions and may cause problems if these variables are accidentally destroyed multiple times. This option is no longer useful on most targets because newer releases of GCC typically put variables into the BSS (the Block Started by Symbol section of an object file, which I always think of as the *Bullshit Section*) without making them common.

`-fconstant-string-class=`*`classname`*: This option, when compiling Objective C programs, tells GCC to use *`classname`* as the name of the class to instantiate for each literal string specified with the syntax `@"..."`. The default *`classname`* is `NXConstantString`.

`-fcprop-registers`: This optimization option tells GCC to perform a copy propagation pass after post-allocation instruction splitting and register allocation to try to reduce scheduling dependencies and eliminate extraneous copies. This option is disabled by default at optimization levels 2 and 3, and when optimizing for size.

`-fcross-jumping`: This optimization option tells GCC to perform a cross-jumping transformation that tries to unify equivalent code and thereby reduce code size. This option is enabled by default at optimization levels 2 and 3, and when optimizing for size.

`-fcse-follow-jumps`: This optimization option, used during CSE, tells GCC to scan through jump instructions when the target of the jump is not reached by any other path. For example, when CSE encounters an `if` statement with an `else` clause, CSE will follow the jump when the condition tested is false.

`-fcse-skip-blocks`: This optimization option, used during common subexpression elimination, is similar to `-fcse-follow-jumps`, but causes CSE to follow jumps that conditionally skip over blocks. When CSE encounters a simple `if` statement with no `else` clause, this option causes CSE to follow the jump around the body of the `if` statement.

`-fcx-limited-range`: This optimization option specifies that a range reduction step should be used when performing complex division. Though this option sets the C99 `CX_LIMITED_RANGE` pragma, it can be used with all GCC compilers.

`-fdata-sections`: This optimization option, for output targets that support arbitrary code sections, causes GCC to place each data item into its own section in the output file. The name of the data item determines the name of the section in the output file. This option is typically used on systems with HP/PA or SPARC processors (under HPUX or Solaris 2, respectively), whose linkers can perform optimizations that improve the locality of reference in the instruction space. These optimizations may also be available on AIX and systems using the ELF object format. Using this option causes the assembler and linker to create larger object and executable files that may therefore be slower on some systems. Using this option also prevents the use of gprof on some systems and may cause problems if you are also compiling with the `-g` option.

`-fdelayed-branch`: This optimization option, if supported by the target machine, causes GCC to attempt to reorder instructions in order to exploit instruction slots that are available after delayed branch instructions.

`-fdelete-null-pointer-checks`: This optimization option tells GCC to use global dataflow analysis to identify and eliminate useless checks for null pointers. The compiler assumes that dereferencing a null pointer would have halted the program, so that pointers that are checked after being dereferenced cannot be null. On systems that can safely dereference null pointers, you can use `-fno-delete-null-pointer-checks` to disable this optimization.

`-fdiagnostics-show-location=[once|every-line]`: This option tells GCC's diagnostic message formatter how frequently diagnostic messages should display source information when diagnostic messages are wrapped across multiple lines. The default value is `once`; specifying *`every-line`* causes the diagnostic formatter to specify the source location on every output line, even if it is just a continuation of a previous message.

`-fdiagnostics-show-options`: This debugging option tells the GCC diagnostics engine to identify the command-line option associated with each diagnostic message that is displayed.

`-fdollars-in-identifiers`: This C++ compilation option tells GCC to accept the dollar sign symbol ($) in identifiers. You can also explicitly prohibit use of dollar signs in identifiers by using the `-fno-dollars-in-identifiers` option. This is a backward-compatibility option; K&R C allows the character $ in identifiers, but ISO C and C++ forbid the use of this character.

`-fdump-class-hierarchy` | `-fdump-class-hierarchy-`*`options`*: These debugging options, when compiling C++ programs, cause GCC to dump a representation of each class's hierarchy and virtual function table layout to a file. The filename is constructed by appending `.class` to the name of each source file. If the *`options`* form is used, the values specified in *`options`* control the details of the dump as described in the definition for the `-fdump-tree` option.

`-fdump-ipa-`*`switch`*: This debugging option controls dumping the language tree to a file at various points of interprocedural analysis, as identified by *switch*. When *switch* is `cgraph`, information about call-graph optimization, unused function removal, and inlining is dumped. Setting *switch* to `all` enables all interprocedural language dumps.

`-fdump-translation-unit` | `-fdump-translation-unit-`*`options`*: These debugging options, when compiling C++ programs, cause GCC to dump a representation of the tree structure for the entire translation unit to a file. The filename is constructed by appending `.tu` to the name of each source file. If the *options* form is used, the values specified in *options* control the details of the dump as described in the description of the `-fdump-tree` options.

`-fdump-tree-`*`switch`* | `-fdump-tree-`*`switch-options`*: These debugging options, when compiling C++ programs, cause GCC to dump the intermediate language tree to a file at various stages of processing. The filename is constructed by appending a *switch*-specific suffix to the source filename, preceded by a `.t`*`XX`*`.` prefix, where *XX* identifies the numeric sequence of the pass made when processing the language tree. The following tree dumps are possible:

- `alias`: Dumps aliasing information for each function.
- `all`: Enables all the available tree dumps with the flags provided in this option. With no flags, this option enables the `cfg`, `generic`, `gimple`, `original`, and `vcg` dumps.
- `ccp`: Dumps each function after CCP (conditional constant propagation).
- `cfg`: Dumps the control flow graph of each function to a file.
- `ch`: Dumps each function after copying loop headers.
- `copyprop`: Dumps trees after copy propagation.
- `copyrename`: Dumps each function after applying the copy rename optimization.
- `dce`: Dumps each function after dead-code elimination.
- `dom`: Dumps each function after applying dominator tree optimizations.
- `dse`: Dumps each function after applying dead-store elimination.
- `forwprop`: Dumps each function after forward propagating single-use variables.
- `fre`: Dumps trees after full redundancy elimination.
- `gimple`: Dumps each function to a file before and after the gimplification pass.
- `inlined`: Dumps after function inlining.
- `mudflap`: Dumps each function after adding mudflap instrumentation.
- `nrv`: Dumps each function after applying the named return value optimization on generic trees.
- `optimized`: Dumps after all tree-based optimizations.
- `original`: Dumps before any tree-based optimization.
- `phiopt`: Dumps each function after optimizing the `phi` operator nodes into straightline code. The `phi` operator is a special operator used to assign different values based on how a block is reached.
- `pre`: Dumps trees after partial redundancy elimination.
- `salias`: Dumps structure aliasing variable information to a file.
- `sink`: Dumps each function after performing code sinking.
- `sra`: Dumps each function after performing scalar replacement of aggregates.

- `ssa`: Dumps SSA-related information to a file.
- `store_copyprop`: Dumps trees after store copy propagation.
- `storeccp`: Dumps each function after store conditional constant propagation.
- `vcg`: Dumps the control flow graph of each function to a file in VCG format.
- `vect`: Dumps each function after applying vectorization of loops.
- `vrp`: Dumps each function after value range propagation.

If the *`options`* form is used, *`options`* is a list of hyphen-separated options that control the details of the dump. Any options that are irrelevant to a specific dump are ignored. The following options are available:

- `address`: Specifying this dump-tree option prints the address of each node and is primarily used to associate a specific dump-tree file with a specific debugging environment.
- `all`: Specifying this dump-tree option turns on all options except `raw`, `slim`, and `lineno`.
- `blocks`: Specifying this dump-tree option prints basic block boundaries and is disabled in raw dumps.
- `details`: Specifying this dump-tree option increases the level of information provided in the dump files where possible.
- `lineno`: Specifying this dump-tree option displays line numbers for statements.
- `raw`: Specifying this dump-tree option prints a raw representation of the dump tree. By default, trees are pretty printed with a C-like representation.
- `slim`: Specifying this dump option prevents dumping members of a scope or function body simply because the end of that scope has been reached and only dumps such items when they are directly reachable by some other path. When dumping pretty-printed trees, this option prevents dumping the bodies of control structures.
- `stats`: Specifying this option dumps various statistics about each available pass.
- `uid`: Specifying this dump option displays the unique ID (DECL_UID) for each variable.
- `vops`: Specifying this option displays the virtual operands for every statement.

`-fdump-unnumbered`: This option, when doing debugging dumps due to the use of the `-d` option, causes GCC to suppress instruction and line numbers. This makes it easier to use `diff` to compare debugging dumps from multiple runs of GCC with different compiler options, most specifically with and without the `-g` option.

`-fearly-inlining`: This debugging option tells GCC to inline specific types of functions early—functions that are marked by `always_inline` and functions whose body seems smaller than the overhead required for a function call. This is done before profiling the code and before the actual inlining pass, and simplifies profiling. This option is enabled by default when doing optimization.

`-feliminate-dwarf2-dups`: This debugging option causes GCC to compress DWARF2 debugging information by eliminating any duplicate information about each symbol but is only meaningful when generating DWARF2 debugging information.

`-feliminate-unused-debug-symbols`: This debugging option causes GCC to only generate debugging information (in STABS format) for symbols that are actually used in the source file(s) that are being compiled.

`-feliminate-unused-debug-types`: This debugging option causes GCC to only generate debugging information for types that are actually used in the source file(s) that are being compiled.

`-femit-class-debug-always`: This debugging option causes GCC to generate debugging information for a C++ class in all object files that use that class. The default is to only generate this information in the object file where the class is first encountered. This option is provided to support debuggers that require that this information be present in each object file.

`-fexceptions`: This code generation option tells GCC to enable exception handling and generates any extra code needed to propagate exceptions. This option is on by default when compiling applications written in languages such as C++ that require exception handling. It is primarily provided for use when compiling code written in languages that do not natively use exception handling, but that must interoperate with C++ exception handlers. For some targets, using this option also causes GCC to generate frame unwind information for all functions, which can substantially increase application size although it does not affect execution.

■ **Note** As an optimization, you can use the `-fno-exceptions` option to disable C++ support for exception handling in older C++ applications that do not use exception handling.

`-fexpensive-optimizations`: This optimization option tells GCC to perform a number of minor optimizations that may require a substantial amount of processing time.

`-fexternal-templates`: This option, when compiling C++ applications, causes GCC to apply `#pragma` interface and `#pragma` implementation to template instantiation. Template instances are emitted or suppressed according to the location of the template definition. The use of this option is deprecated.

`-ffast-math`: This optimization option causes GCC to define the preprocessor macro `__FAST_MATH__` and perform internal math optimizations. This option implies the use of the `-fno-math-errno`, `-funsafe-math-optimizations`, and `-fno-trapping-math` options and activates them if they are not specified. This option should never be used in conjunction with GCC's standard `-O` optimization options, because this can result in incorrect output for programs that depend on the exact implementation of IEEE or ISO rules and specifications for math functions.

`-ffinite-math-only`: Specifying this optimization option causes GCC to allow floating-point optimizations to make the assumption that no checks need to be done to verify that arguments and results are NaN (Not-a-Number) or have infinite values.

`-ffixed-`*reg*: This code generation option tells GCC to treat the register identified by *reg* as a fixed register that is never referred to by generated code other than compiler internals. Registers are machine-specific—those valid for each specific GCC output target are defined in the `REGISTER_NAMES` macro in the machine description macro file.

`-ffloat-store`: This optimization option tells GCC not to store floating-point variables in registers and to inhibit other options that might change whether a floating-point value is taken from a register or memory. Using this option prevents excess precision on machines where floating registers keep more precision than a double floating-point value requires, such as the 68000 (with 68881) and x86 architectures. Additional precision may be undesirable in programs that rely on the precise definition of IEEE floating point. You can compile such programs with this option only if you modify them to store relevant intermediate computations in variables rather than in registers.

`-ffor-scope`: When compiling C++ programs, specifying this option tells GCC to limit the scope of variables declared in a `for-init` statement to the `for` loop, as specified by the C++ standard. If the opposite `-fno-for-scope` option is used, the scope of variables declared in a `for-init` statement extends to the end of the enclosing scope. This was the default behavior of older versions of GNU C++ and many traditional implementations of C++. If neither option is specified, GCC follows the C++ standard.

`-fforce-addr`: This optimization option causes GCC to force memory address constants to be copied into registers before doing arithmetic on them.

`-fforce-mem`: This optimization option causes GCC to force memory operands to be copied into registers before doing arithmetic on them. This may produce better code because it makes all memory references potential common subexpressions that can be reduced. When they are not common subexpressions, instruction combination should eliminate the additional overhead of separate register load. This option is automatically turned on when using the `-O2` optimization option.

`-ffreestanding`: This option, when compiling C applications, tells GCC that compilation takes place in a freestanding environment where the standard library may not be available or exist, such as when compiling an operating system kernel. Using this option implies the use of the `-fno-builtin` option, and is equivalent to using the `-fno-hosted` option.

`-ffriend-injection`: This C++ option tells GCC to inject friend functions into the enclosing namespace such that they are visible outside the scope of the class in which they were declared. This follows the conventions documented in the Annotated C++ Reference Manual and used by versions of GCC prior to 4.1, and is provided for compatibility.

`-ffunction-sections`: This optimization option, for output targets that support arbitrary code sections, causes GCC to place each function into its own section in the output file. The name of the function determines the name of the section in the output file. This option is typically used on systems with HP/PA or SPARC processors (under HPUX or Solaris 2, respectively), whose linkers can perform optimizations that improve the locality of reference in the instruction space. These optimizations may also be available on AIX and systems using the ELF object format. Using this option causes the assembler and linker to create larger object and executable files that may therefore be slower on some systems. Using this option also prevents the use of gprof on some systems and may cause problems if you are also compiling with the `-g` option.

`-fgcse`: This optimization option tells GCC to perform a global common subexpression elimination pass, also performing global constant and copy propagation.

■ **Note** When compiling a program using GCC's computed `gotos` extension, you may get better runtime performance if you disable the global common subexpression elimination pass by specifying the `-fno-gcse` option.

`-fgcse-after-reload`: This optimization option causes GCC to perform an additional load elimination pass after reload in order to clean up redundant spilling.

`-fgcse-las`: This optimization option tells GCC to eliminate redundant loads that come after stores to the same memory location during the global common subexpression elimination pass.

`-fgcse-lm`: This optimization option causes GCC to attempt to optimize load operations during global command subexpression elimination. If a load within a loop is subsequently overwritten by a store operation, GCC will attempt to move the load outside the loop and to only use faster copy/store operations within the loop.

`-fgcse-sm`: This optimization causes GCC to attempt to optimize store operations after global common subexpression elimination. When used in conjunction with the `-fgcse-lm` option, loops containing a load/store sequence will be changed to a load before the loop, and a store after the loop whenever possible.

`-fgnu-runtime`: This option, when compiling Objective C programs, causes GCC to generate object code that is compatible with the standard GNU Objective C runtime. This is the default on most systems.

`-fhosted`: This option, when compiling C programs, tells GCC that the entire standard C library is available during compilation, which is known as a *hosted environment*. Specifying this option implies the `-fbuiltin` option. This option is usually appropriate when compiling any application other than a kernel, unless you want to compile statically linked applications. Using this option is equivalent to using the `-fno-freestanding` option.

`-fif-conversion`: This optimization option tells GCC to use conditional execution to convert conditional jumps into nonbranching equivalents, wherever possible.

`-fif-conversion2`: This optimization option tells GCC to attempt to convert conditional jumps into nonbranching equivalents.

`-finhibit-size-directive`: This code generation option tells GCC not to output a `.size` assembler directive or any other internal information that could cause trouble if the function is split and the two portions subsequently relocated to different locations in memory. This option is typically used only when compiling the `crtstuff.c` character handling routine and should not be necessary in any other case.

`-finline-functions`: This optimization option tells GCC to integrate simple functions into the routines that call them if they are simple enough to do so, based on heuristics. If all calls to a given function are integrated and the function is declared static, no independent assembler code for the function is generated independent of that in the routines that call it.

`-finline-functions-called-once`: This optimization option tells GCC to evaluate all static functions that are only called once for integration into the routine from which they are called, even if they are not marked as inline. When such functions are integrated into the calling routines, no stand-alone assembler code for the function is generated.

`-finline-limit=`*n*: This optimization option tells GCC to modify its internal limit on the size of functions that are explicitly marked as inline with the `inline` keyword or that are defined within the class definition in C++. *N* represents the number of pseudoinstructions (an internal GCC calculation) in such functions, excluding instructions related to parameter handling. The default value of *n* is 600. Increasing this value can result in more inlined code, which may cause increased compilation time and memory consumption. Decreasing this value can improve compilation faster, but may result in slower programs because less code will be inlined. This option can be quite useful for programs such as C++ programs that use recursive templates and therefore can substantially benefit from inlining.

`-finstrument-functions`: This code generation option tells GCC to insert instrumentation calls for function entry and exit. The following profiling functions will be called with the address of the current function and the address from which it was called immediately after entering and immediately before exiting from each function:

```
void __cyg_profile_func_enter (void *this_fn, void *call_site);
void __cyg_profile_func_exit (void *this_fn, void *call_site);
```

The first argument is the address of the start of the current function, which can be looked up in the symbol table. You can specify the `no_instrument_function` attribute for specific functions in order to avoid making profiling calls from them. For more information about profiling and for examples of using profiling functions and attributes, see Chapter 6.

`-fipa-pta`: This optimization option tells GCC to perform interprocedural pointer analysis.

`-fivopts`: This optimization option tells GCC to perform induction variable optimizations (strength reduction, induction variable merging, and induction variable elimination) on trees.

`-fkeep-inline-functions`: This optimization option tells GCC to generate a separate, callable version of each function marked as `inline`, even if all calls to a given function have resulted in inline code. This does not apply to functions marked as `extern inline`.

`-fkeep-static-consts`: This optimization option causes GCC to create and allocate all variables declared `static const`, even if the variables are not referenced. This option is active by default. To force GCC to check whether variables are referenced and to only create them if this is the case, use the `-fno-keep-static-consts` option.

`-fleading-underscore`: This code generation option is provided for compatibility with legacy assembly code, and forces C symbols to be created with leading underscores in object files. This is not completely supported on all GCC output targets.

`-fmath-errno`: This optimization option tells GCC to set the value of `ERRNO` after calling math functions that are executed with a single instruction, such as the `sqrt` function. Programs that rely on IEEE exceptions for math error handling may want to use the `-fno-math-errno` option to provide performance improvements while still maintaining IEEE arithmetic compatibility.

`-fmem-report`: This debugging option causes GCC to display statistics about permanent memory allocation at the end of compilation.

`-fmerge-all-constants`: This optimization option causes GCC to attempt to merge identical constants and variables. This increases the scope of the `-fmerge-constants` option and also tries to merge arrays and variables that are initialized with string and floating-point constants. This may generate nonconformant C and C++ code because these languages require all nonautomatic variables to have distinct locations.

`-fmerge-constants`: This optimization option causes GCC to attempt to merge identical string and floating-point constants across compilation units. This option is on by default during optimization on output targets where the assembler and linker support it.

`-fmessage-length=`*n*: This diagnostic option causes GCC to format diagnostic messages so that they fit on lines of *n* characters or less, inducing line wrapping in messages that are greater than *n* characters. The default value for *n* is 72 characters for C++ messages and 0 for all other front ends supported by GCC. If *n* is 0, no line wrapping will be done.

`-fmodulo-sched`: This optimization option tells GCC to perform swing modulo scheduling, where the instructions in innermost loops can be reordered to improve scheduling performance before the first elimination pass.

`-fmove-loop-invariants`: This optimization option causes GCC to move all invariant computations in loops outside the loop. This option was formerly known as `–fmove-all-movables`.

`-fms-extensions`: This option when compiling C programs tells GCC to accept some nonstandard constructs used in Microsoft Foundation Class (MFC) header files. Specifying this option when compiling C++ programs tells GCC to disable pedantic warnings about MFC constructs such as implicit integers and nonstandard syntax for getting pointers to member functions.

`-fmudflap`: This optimization option when compiling C and C++ applications tells GCC to instrument pointer/array dereferencing operations, standard library string/heap functions, and associated constructs with range and validity tests. The instrumented functions and constructs are assumed to be valid. This instrumentation relies on the libmudflap library, which was first introduced in GCC 4.0, and which is linked with the application at link time if this option is specified.

`-fmudflapir`: This optimization option when compiling C and C++ applications tells GCC to perform the same instrumentation as the `-fmudflap` option, but uses a different version of the libmudflap library in which instrumentation ignores pointer reads. This executes more quickly but can enable erroneous reads to propagate within a program.

`-fmudflapth`: This optimization option when compiling multithreaded C and C++ applications tells GCC to perform the same instrumentation as the `-fmudflap` option but uses a different multithreaded version of the libmudflap library.

`-fnext-runtime`: This option, when compiling Objective C programs, causes GCC to generate output that is compatible with the former NeXT computer system runtime that is used on Darwin and Mac OS X systems.

`-fno-access-control`: This option when compiling C++ programs disables access checking. This option is rarely used and is primarily provided to work around problems in the access control code.

`-fno-asm`: This option affects the keywords that are recognized when compiling C and C++ programs. When compiling ISO C99 programs, this option disables the `asm` and `typeof` keywords. When compiling other C programs, this option also disables the `inline` keyword. When compiling C++ programs, this option only disables the `typeof` keyword. You can still use the keywords `__asm__`, `__inline__`, and `__typeof__` in any C or C++ application. This option is automatically enabled when using the `-ansi` option.

`-fno-branch-count-reg`: This optimization option causes GCC to not use decrement and branch instructions on a count register. Instead, GCC generates a sequence of instructions that decrement a register, compare it against zero, and then branch, based upon the result. This option is only meaningful on architectures such as x86, PowerPC, IA-64, and S/390 that provide decrement and branch instructions.

`-fno-builtin` | `-fno-builtin-`*`FUNCTION`*: These C and Objective C options cause GCC not to recognize built-in functions that do not begin with `__builtin_` as a prefix, which guarantees that you will be able to set breakpoints on function calls and replace the functionality of `FUNCTION` by linking with a different library. This rule is always the case in C++ applications compiled with GCC; to specifically invoke GCC's internal functions, you must call them with the `__builtin_` prefix.

> GCC normally generates special code to handle its built-in functions more efficiently. In some cases, built-in functions may be replaced with inline code that makes it difficult to set breakpoints during debugging or to link with external libraries that provide equivalent functions.

In C and Objective C applications, you can use the less restrictive `-fno-builtin-FUNCTION` option to selectively disable specific built-in functions. This option is ignored if no such built-in function is present. Similarly, you can use the `-fno-builtin` option to disable all built-in functions and then modify your applications to selectively map function calls to the appropriate built-ins, as in the following example:

```
#define strcpy(d, s)    __builtin_strcpy ((d), (s))
```

`-fno-common`: This code generation option when compiling a C language application causes GCC to allocate all global variables in the data section of the object file, even if uninitialized, rather than generating them as common blocks. This option is provided for compatibility with existing systems but has the side effect that if the same variable is declared in multiple separately compiled source modules and is not explicitly declared as `extern`, GCC will display an error message during linking.

`-fno-const-strings`: This C++ option causes GCC to assign the type `char *` to string constants rather than `const char *` as specified in the C++ standard. Specifying this option does not allow you to write to string constants unless you also use the `-fwritable-strings` option. Using this option is deprecated in the GCC 3.x compilers and has been removed in GCC 4.x.

`-fno-cprop-registers`: This optimization option causes GCC to perform an additional copy propagation pass to try to improve scheduling and, where possible, eliminate the register copy entirely. This is done after register allocation and post–register allocation instruction splitting.

`-fno-default-inline`: This C++ optimization option causes GCC not to assume that functions declared within a class scope should be inlined. If you do not specify this option, GCC will automatically inline the functions if you use any optimization level (`-O`, `-O2`, or `-O3`) when compiling C++ applications.

`-fno-defer-pop`: This optimization option causes GCC to pop the arguments to each function call as soon as that function returns. GCC normally lets arguments accumulate on the stack for several function calls and pops them all at once on systems where a pop is necessary after a function call return.

`-fno-elide-constructors`: This option when compiling C++ options causes GCC not to omit creating temporary objects when initializing objects of the same type, as permitted by the C++ standard. Specifying this option causes GCC to explicitly call the copy constructor in all cases.

`-fno-enforce-eh-specs`: This option when compiling C++ applications causes GCC to skip checking for violations of exception specifications at runtime. This option violates the C++ standard but may be useful for reducing code size. The compiler will still optimize based on the exception specifications.

`-fno-for-scope`: This option when compiling C++ applications causes GCC to extend the scope of variables declared inside a `for` loop to the end of the enclosing scope. This is contrary to the C++ standard but was the default in older versions of GCC and most other traditional C++ compilers. If neither this option nor the opposite `-ffor-scope` option is specified, GCC adheres to the standard but displays a warning message for code that might be invalid or assumes the older behavior.

`-fno-function-cse`: This optimization option causes GCC not to put function addresses in registers but to instead make each instruction that calls a constant function contain the explicit address of that function. You may need to use this option if you receive assembly errors when using other optimization options.

`-fno-gnu-keywords`: This option affects the keywords that are recognized when compiling C and C++ programs. When compiling ISO C99 programs, this option disables the `asm` and `typeof` keywords. When compiling other C programs, this option also disables the `inline` keyword. When compiling C++ programs, this option only disables the `typeof` keyword. You can still use the keywords `__asm__`, `__inline__`, and `__typeof__` in any C or C++ application. This option is automatically enabled when using the `-ansi` option.

`-fno-gnu-linker`: This code generation option is used when you are compiling code for linking with non-GNU linkers and suppresses generating global initializations (such as C++ constructors and destructors) in the form used by the GNU linker. Using a non-GNU linker also requires that you use the collect2 program during compilation to make sure that the system linker includes constructors and destructors, which should be done automatically when configuring the GCC distribution.

`-fno-guess-branch-probability`: This optimization option tells GCC to use an empirical model to predict branch probabilities, which may prevent some optimizations. Normally, GCC may use a randomized model to guess branch probabilities when none are available from either profiling feedback (`-fprofile-arcs`) or `__builtin_expect`. This may cause different runs of the compiler on the same program to produce different object code, which may not be desirable for applications such as real-time systems.

`-fno-ident`: This code generation option causes GCC to ignore the `#ident` directive.

`-fno-implement-inlines`: This option when compiling C++ programs tells GCC not to generate out-of-line copies of inline functions based on the `#pragma` implementation. This saves space but will cause linker errors if the code for these functions is not generated inline everywhere these functions are called.

`-fno-implicit-inline-templates`: This option when compiling C++ programs tells GCC not to generate code for implicit instantiations of inline templates. This option is typically used in combination with optimization options to minimize duplicated code.

`-fno-implicit-templates`: This option when compiling C++ programs tells GCC to only emit code for explicit instantiations and not to generate code for noninline templates that are instantiated by use.

`-fno-inline`: This optimization option prevents GCC from expanding any function inline, effectively ignoring the `inline` keyword. During normal optimization, the code for functions identified as inline is automatically inserted at each function call.

`-fno-jump-tables`: This code generation option tells GCC not to use jump tables for handling switch statements and is typically used when generating code for use in a dynamic linker that therefore cannot reference the address of a jump or global offset table.

`-fno-math-errno`: This optimization option prevents GCC from setting `ERRNO` after calling math functions that can be executed as a single instruction, such as `sqrt`. Not setting `ERRNO` reduces the overhead of calling such functions, but `ERRNO` should be set in programs that depend on exact IEEE or ISO compliance for math functions. The `-fno-math-errno` option is therefore not turned on by any of the generic GCC optimization options and must be specified manually.

`-fno-nil-receivers`: This Objective C and Objective C++ language option tells GCC to assume that the receiver of any message dispatched in the current translation unit is not nil. This option is only supported in the NeXT runtime used on Mac OS X 10.3 and greater.

`-fno-nonansi-builtins`: This option when compiling C++ programs disables the use of built-in GCC functions that are not specifically part of the ANSI/ISO C specifications. Such functions include `exit()`, `alloca()`, `bzero()`, `conjf()`, `ffs()`, `index()`, and so on.

`-fno-operator-names`: This option when compiling C++ programs prevents GCC from recognizing keywords such as `and`, `bitand`, `bitor`, `compl`, `not`, `or`, and `xor` as synonyms for the C++ operators that represent those operations.

`-fno-optional-diags`: This option when compiling C++ programs disables optional diagnostics that are not mandated by the C++ standard. In version 3.2.2 of GCC, the only such diagnostic is one generated when a name has multiple meanings within a class. Other such diagnostics may be added in future releases of GCC.

`-fno-peephole | -fno-peephole2`: These optimization options disable different types of machine-specific optimizations. Whether one or both of these options are useful on your system depends on how or if they are implemented in the GCC implementation for your system.

`-fno-rtti`: This option, when compiling C++ programs, tells GCC not to generate information about every class with virtual functions. This information is typically used by C++ runtime type identification features such as `dynamic_cast` and `typeid`. Using this option can save space if you do not explicitly use those aspects of C++—they will still be used by internals such as exception handling, but the appropriate information will only be generated when needed.

`-fno-sched-interblock`: This optimization option tells GCC not to schedule instructions across basic blocks, which is normally done by default when using the `-fschedule-insns` option or optimization options `-O2` or higher.

`-fno-sched-spec`: This optimization option tells GCC not to move nonload instructions, which is normally done by default when using the `-fschedule-insns` option or optimization options `-O2` or higher.

`-fno-signed-bitfields`: This option, when compiling C programs, tells GCC that bit fields are unsigned by default (in other words, when neither `signed` or `unsigned` is present in their declaration). Without specifying this option, bit fields are assumed to be signed in order to be consistent with other basic datatypes. This option is redundant when used in conjunction with the `-traditional` option, which causes all bit fields to be unsigned by default.

`-fno-stack-limit`: This code generation option causes GCC to generate code without an explicit limit on stack size.

`-fno-threadsafe-statics`: This C++ language option tells GCC not to generate the extra code required to use the C++ ABI's thread-safe initialization routines for local statics. This can reduce code size when the generated code does not have to be thread-safe.

`-fno-toplevel-reorder`: This optimization option tells GCC not to reorder any top-level functions, `asm` statements, or variable declarations, and is designed to support existing code that depends on its current ordering.

`-fno-trapping-math`: This optimization option causes GCC to generate code that assumes that floating-point operations cannot generate user-visible traps, which may result in faster operation due to reduced overhead when returning from such functions. If you want to experiment with or perform this type of optimization, this option must be explicitly specified because its use may result in code that is not completely IEEE or ISO compliant. This option is never automatically turned on by any standard GCC optimization option.

`-fno-unsigned-bitfields`: This option, when compiling C programs, tells GCC that bit fields are signed by default (in other words, when neither the `signed` or `unsigned` keyword is present in their declaration). This option should not be used in conjunction with the `-traditional` option, which causes all bit fields to be unsigned by default.

`-fno-verbose-asm`: This code generation option minimizes the number of comments inserted into generated assembly code, and is the GCC default.

`-fno-weak`: This option when compiling C++ programs tells GCC not to use weak symbol support, even if it is provided by the linker—GCC's default action is to use weak symbols when they are available. This option is primarily used for testing at this point and generally should not be used.

`-fno-zero-initialized-in-bss`: This optimization option tells GCC to put variables that are initialized to zero in the data section rather than the BSS. Using the BSS for such variables is GCC's default behavior because this can reduce the size of generated code.

`-fnon-call-exceptions`: This option when compiling languages such as C++ that support exceptions and when runtime support for exception handling is present causes GCC to generate code that allows trap instructions to throw exceptions. This option does not enable exceptions to be thrown from arbitrary signal handlers.

`-fobjc-call-cxx-cdtors`: This Objective C and Objective C++ language option tells GCC to check if the instance variables for any Objective C class are C++ objects with a nontrivial constructor or destructor. If so, GCC synthesizes special instance methods that run nontrivial constructors in the right order (returning self) or nontrivial destructors in the opposite order.

`-fobjc-direct-dispatch`: This Objective C and Objective C++ language option tells GCC to enable fast jumps to the message dispatcher.

`-fobjc-exceptions`: This Objective C and Objective C++ language option tells GCC to enable syntactic support for structure exceptions handling, similar to that provided by C++ and Java.

`-fobjc-gc`: This Objective C and Objective C++ language option enables garbage collection in Objective C and Objective C++ programs.

`-fomit-frame-pointer`: This optimization option tells GCC not to keep the frame pointer in a register for functions that do not require a frame pointer. Besides making an additional register available for other code, this option reduces both code size and the execution path by eliminating the instructions required to save, set up, and restore frame pointers. This option will have no effect on systems where the standard function calling sequence always includes frame pointer allocation and setup. When building GCC for a specific platform, the `FRAME_POINTER_REQUIRED` macro determines whether this option is meaningful on a specific target system.

`-fopenmp`: This code generation option tells GCC to support handling the OpenMP directives `#pragma omp` in C/C++ applications, and `!$omp` in Fortran applications. See the OpenMP API Specification at `http://www.openmp.org` for more information about OpenMP and associated directives.

`-foptimize-register-move`: This optimization option tells GCC to reassign register numbers in simple operations in an attempt to maximize register tying. This option is synonymous with the `-fregmove` option and is automatically enabled when using GCC optimization levels 2 and higher.

`-foptimize-sibling-calls`: This optimization option attempts to optimize sibling and tail recursive calls.

`-fpack-struct`: This code generation option tells GCC to attempt to pack all structure members together without holes, reducing memory use in applications that allocate significant numbers of in-memory data structures. This option is rarely used in applications that make extensive use of system functions that may employ offsets based on the default offsets of fields in data structures. Using this option produces code that is not binary-compatible with code generated without this option.

`-fpcc-struct-return`: This code generation option causes GCC to return short (integer-sized) `struct` and `union` values in memory rather than in registers. This option is typically used when linking object code compiled with GCC with object code produced by other compilers that use this convention.

`-fpeel-loops`: This optimization option tells GCC to simplify loops for which enough information is available to extract the first (or first few) problematic iterations of the loop so that the rest of the loop can be simplified.

`-fpermissive`: This option, when compiling C++ code, causes GCC to downgrade the severity of messages about nonconformant code to warnings, rather than treating them as actual errors. This option has no effect if used with the `-pedantic` option.

`-fPIC`: This code generation option tells GCC to emit position-independent code (PIC) that is suitable for dynamic linking but eliminates limitations on the size of the global offset table. PIC uses a global offset table to hold constant addresses that are resolved when an application is executed. This option is therefore only meaningful on platforms that support dynamic linking and is used when generating code for the 680x0, PowerPC, and SPARC processors. If you are interested in potentially reducing the size of the global offset table, you should use the `-fpic` option instead of this one.

`-fpic`: This code generation option tells GCC to generate position-independent code that is suitable for use in a shared library. This option is only meaningful on target platforms that support dynamic linking and use a machine-specific value (typically 16K or 32K) for the maximum size of the global allocation table for an application. If you see an error message indicating that this option does not work, you should use the `-fPIC` option instead of this one.

■ **Note** Code generated for the IBM RS/6000 is always position independent.

`-fPIE`: This code generation option produces code that is the same as that produced when specifying `-fPIC` but which can only be linked into executables. See the `-pie` option for more information.

`-fpie`: Specifying this code generation option produces code that is the same as that produced when specifying `-fpic`, but which can only be linked into executables. See the `-pie` option for more information.

`-fprefetch-loop-arrays`: This optimization option tells GCC to generate instructions to prefetch memory on platforms that support this. Prefetching memory improves the performance of loops that access large arrays.

`-fpretend-float`: This debugging option is often used when cross-compiling applications and tells GCC to compile code, assuming that both the host and target systems use the same floating-point format. Though this option can cause actual floating constants to be displayed correctly, the instruction sequence will probably still be the same as the one GCC would make when actually running on the target machine.

`-fprofile-arcs`: This debugging and optimization option tells GCC to instrument arcs (potential code paths from one function or procedure to another) during compilation to generate coverage data. This coverage information can subsequently be used by gcov or to enable GCC to perform profile-directed block ordering. Arc coverage information is saved in files with the `.da` (directed arc) extension after each run of an instrumented application. To enable profile-directed block-ordering optimizations, you must compile your application with this option, execute it with a representative data set, and then compile the program again with the same command-line options, adding the `-fbranch-probabilities` option. To use this option during code coverage analysis, you must also use the `-ftest-coverage` option.

`-fprofile-generate`: This optimization option tells GCC to instrument applications such that they produce profiling information that can be used for profile feedback optimization. When used, this option must be specified when compiling and also when linking your code.

`-fprofile-use`: This optimization option tells GCC to enable profile feedback optimization. Specifying this option enables the `-fbranch-probabilities`, `-fpeel-loops`, `-ftracer`, `-funroll-loops`, and `-fvpt` options.

`-fprofile-values`: This optimization option tells GCC to add code to generate data about expression values. When used with the `-fbranch-probabilities` option, this data is used to generate `REG_VALUE_PROFILE` notes that can be used in subsequent optimizations.

`-frandom-seed=`*`STRING`*: This debugging option specifies a seed value that GCC should use when it would otherwise use random numbers, such as when generating symbol names and uniqueifiers in the data files used by gcov and the object code that produces them. If you use this option, you must use a different `STRING` value for each file that you compile.

`-freduce-all-givs`: This optimization option tells GCC to strength reduce all general-induction variables used in loops. Strength reduction is an optimization that uses previous calculations or values to eliminate more expensive calls or calculations. This option is activated by default when any GCC optimization level is used.

`-freg-struct-return`: This option when compiling C or C++ applications causes GCC to generate code that returns `struct` and `union` values in registers whenever possible. By default, GCC uses whichever of the `-fpcc-struct-return` or `-freg-struct-return` options is appropriate for the target system.

`-fregmove`: This optimization option tells GCC to reassign register numbers in order to maximize the amount of register tying, and is synonymous with the `-foptimize-register-move` option. This option is active by default when using GCC optimization level 2 or higher.

`-frename-registers`: This optimization option tells GCC to make use of any unallocated registers in order to attempt to avoid false dependencies in scheduled code. This option is therefore most frequently used on systems with large numbers of registers.

`-freorder-blocks`: This optimization option tells GCC to reorder basic blocks in compiled functions in order to improve code locality and minimize the number of branches taken. This option is enabled at optimization levels 2 and 3.

`-freorder-blocks-and-partition`: This optimization option tells GCC to perform the same reordering as specified by the `-freorder-blocks` option, but to also partition hot and cold basic blocks into separate sections of the assembly and object files in order to improve paging and cache locality.

`-freorder-functions`: This optimization option causes GCC to optimize function placement using profile feedback.

`-freplace-objc-classes`: This Objective C and Objective C++ language option tells GCC to emit a special marker that tells the linker not to statically link the resulting object file, enabling the Mac OS X dynamic loader, dylib, to load the class at runtime. This option is only useful on Mac OS X 10.3 or later, as it is associated with the NeXT runtime's fix-and-continue functionality.

`-frepo`: This option when compiling C++ applications enables automatic template instantiation at link time. Using this option also implies the `-fno-implicit-templates` option.

`-frerun-cse-after-loop`: This optimization option tells GCC to re-run common subexpression elimination after loop optimization has been performed.

`-frerun-loop-opt`: This optimization option tells GCC to run the loop optimizer twice, attempting to immediately capitalize on the results of the first pass.

`-freschedule-modulo-scheduled-loops`: This optimization option tells GCC that it can reschedule loops that have already been modulo scheduled. The `-fno-reschedule-modulo-scheduled-loops` option is used to prevent this rescheduling.

`-frounding-math`: This optimization option tells GCC to disable transformations and optimizations that assume default floating-point rounding behavior, which is round-to-zero for all floating-point-to-integer conversions, and round-to-nearest for all other arithmetic truncations.

`-frtl-abstract-sequences`: This size optimization option tells GCC to look for identical sequences of RTL code that it can turn into pseudoprocedures, replacing the original code with calls to the pseudoprocedure.

`-fsched-spec-load`: This optimization option tells GCC to move some load instructions where this is predicted to improve performance, reduce the execution path, or enable subsequent optimizations. This option is typically used only when you are also using the `-O2`, `-O3`, or `-fschedule-insns` options to schedule before register allocation.

`-fsched-spec-load-dangerous`: This optimization option is slightly more aggressive than the `-fsched-spec-load` option and tells GCC to be even more aggressive in moving load instructions in order to improve performance, reduce the execution path, or enable subsequent optimizations. This option is typically only used when you are also using the `-O2`, `-O3`, or `-fschedule-insns` options to schedule before register allocation.

`-fsched-stalled-insns=`*n*: This optimization option identifies the maximum number (*n*) of instructions (if any) that can be moved from the queue of stalled instructions into the ready list during the second scheduling pass.

`-fsched-stalled-insns-dep=`*n*: This optimization option identifies the number (*n*) of instruction groups (cycles) that will be examined for dependency on a stalled instruction that is a candidate for premature removal from the stalled instruction queue. This option is only meaningful if the `-fsched-stalled-insns` option is specified with a nonzero value.

`-fsched-verbose=`*n*: This debugging option tells GCC the amount of output to print to stderr (or specified dump listing file) during instruction scheduling. When $n > 0$, this option displays the same information as `-dRS`. When $n > 1$, this option also displays basic block probabilities, a detailed ready list, and unit/instruction information. When $n > 2$, this option also displays RTL information about abort points, control flow, and regions. When $n > 4$, this option also displays dependency information.

`-fsched2-use-superblocks`: This optimization option tells GCC to use a superblock scheduling algorithm when scheduling after register allocation. This option should only be used when the `-fschedule-isns2` option is specified, or with optimization level 2 or higher.

`-fsched2-use-traces`: This optimization option tells GCC to use the same algorithm as `-fsched2-use-superblocks`, but to also do code duplication as needed to produce faster, if larger, superblocks and resulting binaries.

`-fschedule-insns`: This optimization option, if supported on the target machine, tells GCC to attempt to reorder instructions to eliminate execution stalls that occur when the data required for an operation is unavailable. This option can be quite useful on systems with slow floating-point or memory load instructions by enabling other instructions to execute until the result of the other instructions are available.

`-fschedule-insns2`: This optimization option is similar to `-fschedule-ins` but tells GCC to perform an additional pass to further optimize instruction scheduling after register allocation has been performed. This option can further improve performance on systems with a relatively small number of registers or where memory load instructions take more than one cycle.

`-fsection-anchors`: This optimization option tells GCC to try to reduce the number of symbolic address calculations by using shared *anchor* symbols to address new entries, reducing the number of GOT (global offset table) entries and accesses.

`-fshared-data`: This code generation option causes GCC to locate data and nonconstant variables in the code that is currently being compiled in shared, rather than private, data. This may be useful on operating systems where shared data is literally sharable between processes running the same program.

`-fshort-double`: This code generation option tells GCC to use the same size when storing double and float data.

`-fshort-enums`: This code generation option tells GCC to minimize the amount of storage allocated to enumerated datatypes, only allocating as many bytes as necessary for the complete range of possible values. In other words, the amount of storage associated to `enum` datatypes will be the smallest integer datatype that provides sufficient space.

`-fshort-wchar`: This C programming language option causes GCC to override the underlying datatype used for `wchar_t` so that it is hardwired to be a short unsigned `int` instead of whatever the default datatype is for the target hardware.

`-fsignaling-nans`: This optimization option tells GCC to compile code that assumes that IEEE Not-a-Number signaling may generate user-visible traps during floating-point operations. This option assumes the `-ftrapping-math` option, and causes the `__SUPPORT_SNAN__` preprocessor macro to be defined.

`-fsigned-bitfields`: This C programming language option controls whether a bit field is signed, or unsigned when this option is unspecified. By default, bit fields are typically signed because it is consistent with basic integer types such as `int`, which are also signed unless the `-traditional` option is also specified on the command line.

`-fsigned-char`: This C programming language option causes GCC to define the `char` datatype as signed, requiring the same amount of storage as `signed char`. This option is equivalent to specifying the `-fno-unsigned-char` command-line option.

`-fsingle-precision-constant`: This optimization option tells GCC to handle floating-point constants as single-precision constants instead of implicitly converting them to double-precision constants.

`-fsplit-ivs-in-unroller`: This optimization option tells GCC that it can express the values of induction variables in later iterations of unrolled loops using the value from the first iteration, which breaks long dependency chains and improves scheduling efficiency.

`-fssa`: This optimization option tells GCC to perform its optimizations using static single assignment (SSA) form. The flow graph for each function is first translated into SSA form, optimizations are done while in that form, and the SSA form is then translated back into a flow graph. This option is not available in GCC 4.x compilers.

`-fssa-ccp`: This optimization option tells GCC to do sparse conditional constant propagation in SSA form. The `-fssa` option must also be specified in order to use this option and, like that option, this option is not available in GCC 4.x compilers.

`-fssa-dce`: This optimization option causes GCC to perform aggressive dead-code elimination in SSA form. The `-fssa` option must also be specified in order to use this option and, like that option, this option is not available in GCC 4.x compilers.

`-fstack-check`: This code generation option tells GCC to add extra code to force the operating system to notice whenever the stack is extended, helping ensure that applications do not accidentally exceed the stack size. (The operating system must still monitor the stack size.) This option is primarily useful in multithreaded environments, where more than one stack is in use. The automatic stack overflow detection provided by most systems in single-stack (single-process) environments is usually sufficient without using this option.

`-fstack-limit-register=`*reg* | `-fstack-limit-symbol=`*SYM*: These code generation options tell GCC to add extra code that ensures the stack does not grow beyond a certain value where *reg* is the name of a register containing the limit, or *SYM* is the address of a symbol containing the limit. A signal is raised if the stack grows beyond that limit. For most targets, the signal is raised before the stack crosses the boundary, so the signal can be caught and handled without taking special precautions. If you are using this option, you should also use the GNU linker to ensure that register names and symbol addresses are calculated and applied correctly.

`-fstack-protector`: This optimization option tells GCC to generate extra code to check for buffer overflows in functions with vulnerable objects.

`-fstack-protector-all`: This optimization option tells GCC to generate extra code to check for buffer overflows in all functions.

`-fstats`: This C++ option tells GCC to display front-end processing statistics once compilation has completed. The GNU C++ (G++) development team uses this information.

`-fstrength-reduce`: This optimization option tells GCC to do loop strength reduction and iteration variable elimination.

`-fstrict-aliasing`: This optimization option tells GCC to use the strictest aliasing rules applicable to the language being compiled. For C (and C++), this performs optimizations based on expression type. Objects of two different types are assumed to be located at different addresses unless the types are structurally similar. For example, an `unsigned int` can be an alias for an `int`, but not for a `void *` or `double`. A character type can be an alias for any other type. This option can help detect aliasing errors in potentially complex datatypes such as unions.

`-fsyntax-only`: This diagnostic option tells GCC to check the code for syntax errors, without actually compiling any part of it.

`-ftemplate-depth-`*n*: This C++ option sets the maximum instantiation depth for template classes to *n*. Limits on template instantiation depth are used to detect infinite recursion when instantiating template classes. ANSI/ISO C++ standards limit instantiation depth to 17.

`-ftest-coverage`: This debugging option causes GCC to create two data files for use by the gcov code-coverage utility. The first of these files is source.bb, which provides a mapping of basic blocks to line numbers in the source code. The second of these files is source.bbg, which contains a list of all of the arcs in the program's flow graph. If this option is used with the `-fprofile-arcs` option, executing the compiled program will also create the data file source.da, which provides runtime execution counts used in conjunction with the information in the source.bbg file. Coverage data generally maps better to source files if no optimization options are used when generating code coverage information.

`-fthread-jumps`: This optimization option optimizes jumps that subsequently perform redundant comparisons, skipping those comparisons, and redirecting the code to the appropriate point later in the code execution flow.

`-ftime-report`: This debugging option causes GCC to display statistics about the time spent in each compilation pass.

`-ftls-model=`*`MODEL`*: This code generation option tells GCC to use a specific thread-local storage model. *`MODEL`* should be either `global-dynamic`, `local-dynamic`, `initial-exec`, or `local-exec`. The default is `global-dynamic`, unless the `-fpic` option is used, in which case the default is `initial-exec`.

`-ftracer`: This code generation pass simplifies the control flow of functions, allowing other optimizations to do a better job.

`-ftrapv`: This optimization option causes GCC to generate traps for signed overflow on addition, subtraction, and multiplication operations.

`-ftree-ccp`: This optimization option tells GCC to perform sparse conditional constant propagation on trees, and is enabled at optimization level 1 and higher.

`-ftree-ch`: This optimization option tells GCC to perform loop header copying on trees, which can improve the effectiveness of code motion optimization and saves a jump. This option is enabled at all optimization levels except for `-Os`, because it can increase code size.

`-ftree-copy-prop`: This optimization option tells GCC to perform copy propagation on trees, eliminating unnecessary copies. This option is enabled at all optimization levels.

`-ftree-copyrename`: This optimization option tells GCC to perform copy renaming on trees and is active at all optimization levels.

`-ftree-dce`: This optimization option tells GCC to perform dead-code elimination on trees and is enabled at all optimization levels.

`-ftree-dominator-opts`: This optimization option tells GCC to perform a variety of simple scalar cleanups based on a denominator tree traversal, including constant/copy propagation, expression simplification, redundancy elimination, and range propagation, and also performs jump threading, reducing jumps to jumps. This option is enabled at all optimization levels.

`-ftree-dse`: This optimization option tells GCC to perform dead store elimination on trees, and is enabled at all optimization levels.

`-ftree-fre`: This optimization option tells GCC to perform full redundancy elimination on trees, which only considers expressions that are computed on all paths to the redundant computation. This option is enabled at all optimization levels.

`-ftree-loop-im`: This optimization option tells GCC to perform loop invariant motion on trees, moving the operands of invariant conditions out of loops and moving invariants that would be hard to handle at the RTL level, such as function calls and operations that expand into nontrivial numbers of instructions.

`-ftree-loop-ivcanon`: This optimization option tells GCC to create a variable to track the number of loop iterations, which can be useful in subsequent loop unrolling.

`-ftree-loop-linear`: This optimization option tells GCC to perform linear loop transformations on trees, which can improve cache performance and enable further loop optimizations.

`-ftree-loop-optimize`: This optimization option tells GCC to perform loop optimization on trees and is enabled at all optimization levels.

`-ftree-lrs`: This optimization option tells GCC to perform live range splitting in SSA to normal translations and is enabled at all optimization levels.

`-ftree-pre`: This optimization option tells GCC to perform partial redundancy elimination on trees and is enabled at optimization levels 2 and 3.

`-ftree-salias`: This optimization option tells GCC to perform structure alias analysis on trees and is enabled at all optimization levels.

`-ftree-sink`: This optimization option tells GCC to perform forward store motion on trees and is enabled at all optimization levels.

`-ftree-sra`: This optimization option tells GCC to perform the scalar replacement of aggregates and is enabled at all optimization levels.

`-ftree-store-ccp`: This optimization option tells GCC to perform sparse conditional constant propagation on trees and is enabled at all optimization levels.

`-ftree-store-copy-prop`: This optimization option tells GCC to perform copy propagation of memory loads and stores and is enabled at optimization levels 2 and 3.

`-ftree-ter`: This optimization option tells GCC to perform temporary expression replacement in SSA to normal translations and is enabled at all optimization levels.

`-ftree-vect-loop-version`: This optimization option tells GCC to do loop versioning when doing loop vectorization, generating both vectorized and nonvectorized versions of the loop that are selected at runtime based on checks for alignment or dependencies. This option is enabled at all optimization levels except for `-Os`.

`-ftree-vectorize`: This optimization option tells GCC to perform loop vectorization on trees and is enabled at all optimization levels.

`-ftree-vectorizer-verbose=`*n*: Specifying this debugging option controls the amount of debugging information produced during loop vectorization. When $n = 0$, no output is produced. Higher values of *n* produce increasing amounts of information about loops that were vectorized, loops that were considered for vectorization, and loops that were not vectorized.

`-funroll-all-loops`: This optimization causes GCC to unroll all loops, even if the number of times they are executed cannot be guaranteed when the loop is entered. Though this usually makes programs run more slowly, it provides opportunities for subsequent optimization through code elimination.

`-funroll-loops`: This optimization causes GCC to unroll loops, but limits the loops that will be unrolled to those where the number of times they are executed can be determined at compile time or when entering the loop. Using this option implies both the `-fstrength-reduce` and `-frerun-cse-after-loop` options. Using this option makes code larger but does not guarantee improved performance or execution speed. It does provide opportunities for subsequent optimization through other GCC optimization options.

`-funsafe-loop-optimizations`: This optimization option tells GCC to enable additional loop optimizations by assuming that all loop indices are valid and that no loops with nontrivial exit conditions are not infinite.

`-funsafe-math-optimizations`: This optimization option enables GCC to perform optimizations for floating-point arithmetic that assumes that arguments and results are valid and may violate IEEE or ANSI standards. This option should never be used in conjunction with any standard optimization (`-O`) option because it can result in incorrect output in programs that depend on an exact implementation of the IEEE or ISO specifications for math functions.

`-funsigned-bitfields`: This C language option controls whether a bit field is signed or unsigned when neither keyword is specified. Bit fields are ordinarily signed by default because this is consistent with basic integer types such as `int`, which are signed types. Using the `-traditional` option forces all bit fields to be unsigned regardless of whether this option is specified.

`-funsigned-char`: This C language option forces the `char` datatype to be unsigned. This overrides any default character datatype defaults for a given system. This option is valuable when porting code between system types that have different defaults for the `char` datatype. Note that the `char` type is always distinct from `signed char` and `unsigned char` even though its behavior is always the same as either of those two.

`-funswitch-loops`: This optimization option tells GCC to move branches with loop invariant conditions outside loops, providing duplicates of the loop based on the condition result on both branches.

`-funwind-tables`: This code generation option tells GCC to enable exception handling and generates any static data used when propagating exceptions. This option is similar to the `-fexceptions` option, but generates static data rather than code. This option is rarely used from the command line and is usually incorporated into language processors that require this behavior (such as C++).

`-fuse-cxa-atexit`: This C++ option causes GCC to register destructors for objects with static storage duration using the `__cxa_atexit()` function rather than the `atexit` function. This option is required for fully standards-compliant handling of static destructors but only works on systems where the C library supports the `__cxa_atexit()` function.

`-fvar-tracking`: This debugging option tells GCC to run an additional variable tracking pass that computes where variables are stored at each code position and therefore provides better debugging information. If supported by the debug information format, this option is enabled when compiling for any optimization level and when generating debug information using `-g`.

`-fvariable-expansion-in-unroller`: This optimization option tells GCC to create multiple copies of local variables during loop unrolling.

`-fverbose-asm`: This code generation option inserts extra comments into generated assembly code to make it more readable. This option is generally only used during manual optimization or by people who are verifying the generated assembly code (such as the GCC maintenance and development teams).

`-fvisibility=`*VALUE*: This code generation option tells GCC to set the default ELF symbol visibility to `VALUE`, where *VALUE* is one of the standard C++ values `DEFAULT`, `INTERNAL`, `HIDDEN`, or `PROTECTED`. The default value is `DEFAULT` (public), which makes all symbols visible. See the discussion of visibility attributes in Chapter 2 for alternate ways of specifying symbol visibility through attributes.

`-fvisibility-inlines-hidden`: This C++ language option causes all inlined methods to be marked with `__attribute__ ((visibility ("hidden")))` so that they do not appear in the export table of shared objects, which can therefore significantly improve shared object load time.

`-fvolatile`: This code generation option tells GCC to consider all memory references through pointers to be volatile.

`-fvolatile-global`: This code generation option tells GCC to consider all memory references to `extern` and global data items to be volatile. This option does not cause GCC to consider static data items to be volatile.

`-fvolatile-static`: This code generation option tells GCC to consider all memory references to static data to be volatile.

`-fvpt`: This optimization option when used with the `-fprofile-arcs` option tells GCC to add code to collect information about expression values that can be used by the `-fbranch-probabilities` option during subsequent optimizations.

`-fvtable-gc`: This C++ option tells GCC to generate special relocations for vtables and virtual function references. This enables the linker to identify unused virtual functions and zero out vtable slots that refer to them. This option is commonly used with the `-ffunction-sections` option and the linker's `-Wl,--gc-sections` option, in order to also discard the functions themselves. This optimization requires that you are also using GNU as and GNU ld, and is not supported on all system types. Note that the `-Wl,--gc-sections` option is ignored unless the `-static` option is also specified.

`-fweb`: This optimization option tells GCC to construct the webs used for register allocation purposes and assign each web to an individual pseudoregister. This enables the register allocation pass to operate on these pseudoregisters and also strengthens other optimization passes, such as common subexpression elimination, loop optimization, and dead-code removal. This option is enabled by default when `-funroll-loops` is specified.

`-fwhole-program`: This optimization option tells GCC to assume that the current compilation unit represents the entire program, enabling more aggressive optimization because all functions (except `main`) and variables can be assumed to be static.

`-fworking-directory`: This preprocessor option tells GCC to generate line numbers in preprocessor output that identify the current working directory at compile time. This option is enabled when any debugging information is being generated.

`-fwrapv`: This code generation option tells GCC to assume that signed arithmetic overflow wraps around using two's-complement representation. This option is enabled by default for GCC's Java compiler, as required by the Java specification.

`-fwritable-strings`: This C language option tells GCC to store string constants in the writable data segment without uniquifying them. This option is provided for compatibility with older programs that assume they can write into string constants (even though this is poor form). Specifying the `-traditional` option also causes this behavior. This option is not supported in the GCC 4.x compilers.

`-fzero-link`: This Objective C and Objective C++ language option tells GCC to preserve calls of `objc_getClass()` that would ordinarily be replaced by static references that are initialized at load time. This option is only useful when compiling for the Mac OS X platform.

`-g`: This debugging option causes GCC to include debugging and symbol table information in object files, which can subsequently be used by GDB. The format of these object files and the debugging information that they contain depends on the native binary format associated with the platform, and can be one of the following: COFF (SVR3 systems), DWARF (SVR4), STABS (Linux), or XCOFF (AIX).

> On most systems that use STABS format, including the `-g` option enables the use of extra debugging information that only GDB can employ; this extra information makes debugging work better in GDB but will probably make other debuggers crash or refuse to read the program. If you want to ensure generation of the extra information, use `-gstabs+`, `-gstabs`, `-gxcoff+`, `-gxcoff`, `-gdwarf-1+`, `-gdwarf-1`, or `-gvms`. Users of the `-gvms` option have my inherent sympathy.
>
> Unlike most other C compilers, GCC allows you to use `-g` with optimization options such as `-O`, but this is not recommended. As you would expect, optimized code may occasionally produce surprising results due to optimization in loop structure, loop control, variable elimination, statement movement, result compression, and so on.
>
> Other than the format-related options listed previously, related debugging options include `-ggdb`, `-glevel` (and similar level options), profiling options such as `-p`, `-pg`, `-Q`, `-ftime-report`, `-fmem-report`, `-fprofile-arcs`, `-ftest-coverage`, `-dletters`, `-fdump-unnumbered`, `-fdump-translation-unit`, `-fdump-class-hierarchy`, `-fdump-tree`, `-fpretend-float`, `-print-multi-lib`, `-print-prog-name=program`, `-print-libgcc-file-name`, `-print-file-name=libgcc.a`, `-print-search-dirs`, `-dumpmachine`, `-dumpversion`, and `-dumpspecs`.

`-g`*`LEVEL`* | `-ggdb`*`LEVEL`* | `-gstabs`*`LEVEL`* | `-gcoff`*`LEVEL`* | `-gxcoff`*`LEVEL`* | `-gvms`*`LEVEL`*: These debugging options cause GCC to produce debugging information in various formats, using the `LEVEL` value to specify the amount of information displayed. The default value of `LEVEL` is 2. Level 1 produces minimal information, sufficient for displaying backtraces in parts of the program that the user does not expect to debug. Level 3 includes information such as the macro definitions present in the program.

`-gcoff`: This causes GCC to produce debugging information in the COFF format used by SDB on most System V systems prior to System V Release 4.

`-gdwarf`: This option causes GCC to produce debugging information in DWARF (version 1) format, used by the SDB debugger on most System V Release 4 systems.

`-gdwarf+`: This option causes GCC to produce debugging information in DWARF version 1 format using GNU extensions understood only by GDB. Debugging applications compiled with this option using other debuggers, or compiling them with toolchains that are not strictly composed on GNU tools may cause unexpected behavior.

`-gdwarf-2`: This option causes GCC to produce debugging information in DWARF version 2 format, which is used by the DBX debugger on IRIX 6.

`-gen-decls`: This Objective C option causes GCC to dump interface declarations for all classes seen in each source file to a file named sourcename.decl.

`-ggdb`: This option causes GCC to produce debugging information that is targeted for use by GDB. Using this option tells GCC to employ the debugging options that provide the greatest amount of detail for the target platform.

`-gstabs`: This option forces GCC to produce debugging information in the STABS format without using the GDB extensions. The STABS format is used by most debuggers on BSD and Linux systems. GDB can still be used to debug these applications but will not be able to debug them as elegantly as if the `-gstabs+` option had been specified.

`-gstabs+`: This option causes GCC to produce debugging information in STABS format, using the GDB extensions that are only meaningful to GDB. Debugging applications compiled with this option, using other debuggers or compiling them with toolchains that are not strictly composed on GNU tools may cause unexpected behavior.

`-gvms`: This sad, lonely debugging option is only relevant to users who are writing applications intended to be executed and debugged by the DEBUG application on VMS systems. `SYS$SYSOUT`, anyone?

`-gxcoff`: This option causes GCC to produce debugging information in XCOFF format, if supported on the execution platform. XCOFF is the format used by the classic Unix DBX debugger on IBM RS/6000 systems.

`-gxcoff+`: This option causes GCC to produce debugging information in XCOFF format, using debugging extensions that are only understood by GDB. Debugging applications compiled with this option using other debuggers or compiling them with toolchains that are not strictly composed on GNU tools may cause unexpected behavior.

`-H`: This option causes the preprocessor to print the name of each header file used during preprocessing. Each name is indented to show how deeply nested that `#include` statement is.

`--help`: This output option causes GCC to display a summary list of the command-line options that can be used with GCC. Using this option in conjunction with the `-v` option causes GCC to also pass the `--help` option to all subsequent applications invoked by GCC, such as the assembler, linker, and loader, which will also cause them to display a list of many of the standard command-line options that they accept. Adding the `-W` command-line option to the `--help` and `-v` options will also cause GCC and subsequent applications in the toolchain to display command-line options that can be used but have no documentation except for in this book.

`-I-`: This directory search option is used during preprocessing to identify additional directories that should be searched for the #include definition files used in C and C++ applications. Any directories specified using the `-I` option before the `-I-` option are only searched for files referenced as #include "file", not for files referenced as #include <file>. If you use the `-I` option to specify additional directories to search after using the `-I-` option, those additional directories will be searched both for files referenced as #include "file" and #include <file>.

> Using the `-I-` option also keeps the preprocessor from examining the working directory for #include files referenced in #include statements. You can subsequently use the `-I` option to explicitly search the working directory if you specify it by name on the command line.

■ **Note** Using the `-I-` option does not cause GCC to ignore the standard system directories in which #include files are typically located. To do this, specify the `-nostdinc` option.

`-IDIR`: This directory search option is used during preprocessing to identify additional directories that should be added to the beginning of the list of directories that are searched for #include files. This can be used to override #include files in an operating system's include directories that have the same name as #include files local to your application's source code. If you use more than one `-I` option on the GCC command line, the specified directories are scanned in the order that they are specified, from left to right, followed by the standard system directories.

■ **Note** You should not use this option to add #include directories that contain vendor-supplied system header files. Such directories should be specified using the `-isystem` option, which puts them in the search order after directories specified using the `-I` option, but before the system header files.

■ **Caution** Using the `-I` option to specify system include directories (such as `/usr/include` and `/usr/include/sys`) is not a good idea. GCC's installation procedure often corrects bugs in system header files by copying them to GCC's include directory and then modifying its copies. Because GCC is your friend, it will display a warning whenever you specify a system include directory using this option.

`-imacros file`: This preprocessor option causes cpp to load all macros defined in the specified file, but to discard any other output produced by scanning the file. All files specified by the `-imacros` options are processed before any files specified by using the `-include` option. This enables cpp to acquire all of the macros defined in the specified file without including any other definitions that it contains.

`-imultilib dir`: This preprocessor option tells GCC to use `dir` as a subdirectory of the directory that contains target-specific C++ headers.

`-include file`: This preprocessor option causes cpp to process the specified file as if it were specified using a `#include "file"` statement in the first line of the primary source file. The first directory searched for *file* is the preprocessor's working directory, which may differ from the directory containing the main source file. If the specified file is not found there, cpp searches for it through the remainder of the normal include file search chain. If the `-include` option is specified multiple times, the specified files are included in the order that they appear on the command line.

`-iprefix prefix`: This preprocessor option causes cpp to use `prefix` as the prefix for subsequent `-iwithprefix` options. If the specified prefix is a directory, it should end with a trailing `/`.

`-iquote dir`: This directory option tells GCC to search `dir` for header files specified using `#include "file"` before searching all directories specified using `-I` and before all system directories.

`-isysroot dir`: This preprocessor option tells GCC to use `dir` as the logical root directory for all searches for header/include files.

`-isystem dir`: This preprocessor option causes cpp to search `dir` for header files after all directories specified by the `-I` option have been searched, but before the standard system directories. The specified directory is also treated as a system include directory.

`-iwithprefix dir|-iwithprefixbefore dir`: These preprocessor options cause cpp to append the specified `dir` to any prefix previously specified with the `-iprefix` option, and to add the resulting directory to cpp's search path for include directories. Using the `-iwithprefix` option adds this directory to the beginning of cpp's search path (just as the `-I` option would). Using the `-iwithprefixbefore` option adds this directory to the end of cpp's search path (just as the `-idirafter` option would).

`-L`*`dir`*: This directory search option causes the linker to add the specified directory `dir` to the list of directories to be searched for libraries specified using the `-l` command-line option.

`-l`*`LIBRARY`*`|-l` *`LIBRARY`*: These linker options cause the linker to search the library with the base library name of *LIBRARY* (i.e., the full name of lib*LIBRARY*.a) when linking. Object files and libraries are searched based on the order in which they are specified on the linker command line. The only difference between these options and explicitly specifying the name of the library is that these options surround *LIBRARY* with `lib` and `.a`, and search for the specified library in multiple directories.

`-M`: This preprocessor option causes cpp to generate a rule suitable for use by the make program that describes the dependencies of the main source file, rather than actually preprocessing the source files. This make rule contains the name of that source file's corresponding output file, a colon, and the names of all included files, including those coming from all `-include` or `-imacros` command-line options. Unless explicitly specified using the `-MT` or `-MQ` command-line options, the name of the object file is derived from the name of the source file in the standard fashion, by replacing any existing extension with the `.o` extension associated with object files. If the rule is extremely long, it is broken into multiple lines whose new lines are escaped using the backslash character (`\`).

> You can use the `-M` preprocessor option with the `-MF` option to specify the output file, which is recommended if you are also using debugging options such as `-dM` to generate debugging output. Specifying this option also invokes the `-E` option, automatically defining the `__GNUC__`, `__GNUC_MINOR__`, and `__GNUC_PATCHLEVEL__` macros and causing the compilation process to stop after the preprocessing phase.

`-MD`: This preprocessor option is equivalent to specifying the `-M -MF file` options, except that the `__GNUC__`, `__GNUC_MINOR__`, and `__GNUC_PATCHLEVEL__` macros are not defined and that compilation continues after the preprocessing stage. Since this option does not take an output `file` argument, cpp first checks if the name of an output file has been specified using the `-o` option. If so, the output file is created using the basename of that file and replacing any existing extension with the `.d` extension. If not, the name of the output file is derived from the name of the input file, again replacing any existing suffix with the `.d` suffix. Because the `-MD` option does not imply the `-E` option, this option can be used to generate a dependency make rule output file as part of the complete compilation process.

■ **Note** If the `-MD` option is used in conjunction with the `-E` option, any `-o` option specifies the name of the dependency output file. If used without the `-E` option, the `-o` option specifies the name of the final object file.

`-MF file`: This preprocessor option along with the `-M` or `-MM` options identifies the name of a file to which cpp should write dependency information. If the `-MF` option is not specified, the dependency rules are sent to the place to which preprocessor output would have been sent.

`-MG`: This preprocessor option along with the `-M` or `-MM` options causes cpp to treat missing header files as generated files that should be located in the same directory as the source file. This option also suppresses generating preprocessed output, because a missing header file is considered an error. This option is often used when automatically updating Makefiles.

`-MM`: This preprocessor option causes GCC to generate the same output rule as that produced by the `-M` option, the difference being that the generated rules do not list include files that are found in system include directories, or include files that are included from system include files.

`-MMD`: This preprocessor option causes GCC to generate the same output rule as that produced by the `-M` option, the difference being that the generated rules do not list user include files or other user include files that are included by user include files.

`-MP`: This preprocessor option causes cpp to add a fake target for each dependency other than the main file, causing each to depend on nothing through a dummy rule. These rules work around errors that may be generated by the make program if you remove header files without making corresponding Makefile updates.

`-MQ target`: This preprocessor option causes cpp to change the output target in the rule emitted by dependency-rule generation. Instead of following the standard extension-substitution naming convention, using this option sets the name of the output file to the name that you specify. Any characters that have special meaning to the make program are automatically quoted.

`-MT target`: This preprocessor option causes cpp to change the output target in the rule emitted by dependency-rule generation. Instead of following the standard extension-substitution naming convention, using this option sets the name of the output file to the exact filename that you specify.

`-no-integrated-cpp`: This debugging option for C and C++ applications causes GCC to invoke the external cpp rather than the internal C preprocessor that is included with GCC. The default is to use the internal cpp that is provided with GCC. Specifying this option in conjunction with the `-B` option enables you to integrate a custom C preprocessor into your GNU toolchain and allows you to integrate a user-supplied cpp that you then specify via the `-B` option. For example, you could name your preprocessor mycpp and then cause GCC to use it by specifying the `-no-integrated-cpp -Bmy` option sequence.

`-nodefaultlibs`: This linker option tells GCC not to use the standard system libraries when linking.

`-nostartfiles`: This linker option tells GCC not to use the standard system start files (crt0.o, etc.) when linking.

`-nostdinc`: This C language directory search option prevents the preprocessor from searching the standard system directories for #include files specified using `#include <file>` statements.

`-nostdinc++`: This C++ language directory search option prevents the preprocessor from searching the C++-specific system directories for #include files specified using `#include <file>` statements. Standard system include directories such as `/usr/include` and `/usr/include/sys` are still searched.

`-nostdlib`: This linker option tells GCC not to use either the standard system libraries or the startup files when linking.

`-O | -O1`: These optimization options cause GCC to attempt to reduce the size and improve the performance of the target application. On most systems, the `-O` option turns on the `-fthread-jumps` and `-fdelayed-branch` options.

> Without optimization, GCC's primary goal is to compile applications as quickly as possible. A secondary goal is to make it easy to subsequently debug those applications if necessary. Compiling with optimization will almost certainly take more time and will also require more memory when compiling any sizable function or module. Optimization may also combine variables or modify the execution sequence of an application, which can make it difficult to debug an optimized application. You rarely want to specify an optimization option when compiling an application for debugging unless you are debugging the optimization process itself.

`-O0`: This optimization option explicitly disables optimization. This option is the equivalent of not specifying any `-O` option. While seemingly meaningless, this option is often used in complex Makefiles where the optimization level is specified in an environment variable or command-line Makefile option.

`-O2`: This optimization option causes GCC to attempt additional optimizations beyond those performed for optimization level 1. In this optimization level, GCC attempts all supported optimizations that do not trade off between size and performance. This includes all optimization options with the exception of loop unrolling (`-funroll-loops`), function inlining (`-finline-functions`), and register renaming (`-frename-registers`). As you would expect, using the `-O2` option increases both compilation time and the performance of compiled applications.

`-O3`: This optimization option causes GCC to attempt all performance optimizations, even if they may result in a larger compiled application. This includes all optimization options performed at optimization levels 1 and 2, plus loop unrolling (`-funroll-loops`), function inlining (`-finline-functions`), and register renaming (`-frename-registers`).

`-Os`: This optimization option tells GCC to optimize the resulting object code and binary for size rather than for performance.

`-o` *`file`*: This output option tells GCC to write its output binary to the file *file*. This is independent of the type of output that is being produced: preprocessed C code, assembler output, and object module, or a final executable. If the `-o` option is not specified, executable output will be written to the following files:

- *Executables*: Written to a file named a.out (regardless of whether a.out is the execution format)
- *Object files*: Written to files with the input suffix replaced with `.o` (*file*.c output is written to *file*.o)
- *Assembler output*: Written to files with the input suffix replaced with `.s` (for example, assembler output for the file *file*.c output is written to *file*.s)
- *Preprocessed C source code*: Written to standard output

`-P`: This preprocessor option causes the preprocessor to inhibit the generation of line markers in its output and is usually used when the output from the preprocessor will be used with a program that may not understand the line markers.

`-p`: This debugging option causes GCC to generate extra code that will produce profiling information that is suitable for the analysis program, prof. This option must be used both when compiling and linking the source file(s) that you want to obtain profiling information about.

`--param` *`NAME=VALUE`*: This optimization option provides control over the parameters used to control various optimization options. For example, GCC will not inline functions that contain more than a certain number of instructions. The `--param` command-line option gives you fine-grained control over limits such as this. All of these parameters are integer values.

Possible parameters that you can specify are the following:

- `max-delay-slot-insn-search`: The maximum number of instructions to consider when looking for an instruction to fill a delay slot. Increasing values mean more aggressive optimization, resulting in increased compilation time with a potentially small improvement in performance.
- `max-delay-slot-live-search`: The maximum number of instructions to consider while searching for a block with valid live register information when trying to fill delay slots. Increasing this value means more aggressive optimization, resulting in increased compilation time.
- `max-gcse-memory`: The approximate maximum amount of memory that will be allocated in order to perform global common subexpression elimination optimization. If more memory than the specified amount is required, global common subexpression optimization will not be done.
- `max-gcse-passes`: The maximum number of passes of global common subexpression elimination to run.
- `max-inline-insns`: Functions containing more than this number of instructions will not be inlined. This option is functionally equivalent to using the `-finline-limit` option with the same value.
- `max-pending-list-length`: The maximum number of pending dependencies scheduling will allow before flushing the current state and starting over. Large functions with few branches or calls can create excessively large lists that needlessly consume memory and resources.

`-pass-exit-codes`: This output option causes GCC to return the numerically highest error code produced during any phase of the compilation process. GCC typically exists with a standard Unix error code of 1 if an error is encountered in any phase of the compilation.

`-pedantic`: This diagnostic/warning option causes GCC to display all warnings demanded for strict ISO C and ISO C++ compliance. Using this option does not verify ISO compliance, because it only issues warnings for constructs for which ISO C and ISO C++ require such a message (plus some that have been added in GCC but are not strictly mandatory for ISO C/C++). This option also causes GCC to refuse to compile any program that uses extensions and C/C++ syntax that are not ISO compliant. Valid ISO C and ISO C++ programs should compile properly with or without this option (though some may require additional restrictive options such as `-ansi` or an `-std` option specifying a specific version of ISO C).

> Using the `-pedantic` option does not generate warning messages for alternate keywords whose names begin and end with __, and are also disabled for expressions that follow these keywords. These extensions are typically only used in system software, rather than application software.

`-pedantic-errors`: This diagnostic/warning message causes GCC to display error messages rather than warnings for non–ISO C/C++ constructs.

`-pg`: This debugging option causes GCC to generate extra code that produces profile information suitable for use by the analysis program gprof. This option must be used both when compiling and linking the source file(s) that you want to obtain profiling information about.

`-pie`: This linker option tells GCC to produce a position-independent executable for targets that support that type of executable. When using this option for final linking, you must specify the same code generation options as were used to generate each object module.

`-pipe`: This output option causes GCC to use pipes rather than temporary files when exchanging data between various stages of compilation. This can cause problems on systems where non-GNU tools (such as a custom preprocessor, assembler, and so on) are used as part of the compilation toolchain.

`-print-file-name=`*`LIBRARY`*: This debugging option causes GCC to display the full pathname of the specified library. This option is often used when you are not linking against the standard or default system libraries, but you do want to link with a specific library, such as libgcc.a, as in the following example:

```
gcc -nodefaultlibs foo.c bar.c... 'gcc -print-file-name=libgcc.a'
```

When used in this form, surrounding the command `gcc -print-file-name=libgcc.a` with backquotes causes the command to be executed and its output displayed, which is then incorporated on the compilation command line as an explicit reference to the target library.

■ **Note** If the specified library is not found, GCC simply echoes the library name.

`-print-libgcc-file-name`: This debugging option is a shortcut for using the option `-print-file-name=libgcc.a`, and is used in the same circumstances.

`-print-multi-directory`: This debugging option causes GCC to print the directory name corresponding to the multilib selected by any other switches that are given on the command line. This directory is supposed to exist in the directory defined by the `GCC_EXEC_PREFIX` environment variable.

`-print-multi-lib`: This debugging option causes GCC to display the mapping from multilib directory names to compiler switches that enable them. This information is extracted from the specification files used by the compiler, in which the directory name is separated from the switches by a semicolon, and each switch starts with an @ symbol instead of the traditional dash/minus symbol, with no spaces between multiple switches.

WHAT ARE MULTILIBS?

Multilibs are libraries that are built multiple times, each with a different permutation of available machine-specific compiler flags. This makes it easy for GCC to produce output targeted for multiple platforms and to take advantage of different types of optimizations for similar but different platforms by having precompiled versions of libraries targeted for each.

Multilibs are typically built when you are using GCC with multiple targets for a given architecture where you need to support different machine-specific flags for various combinations of targets, architectures, subtargets, subarchitectures, CPU variants, special instructions, and so on.

You can display any multilibs available on your system by executing the `gcc -print-multi-lib` command. Multilibs are specified in entries in the compiler specification files that are stored in the install-dir/lib/architecture/version/specs file associated with each GCC installation.

`-print-prog-name=`*`PROGRAM`*: This debugging option causes GCC to display the full pathname of the specified program, which is usually a part of the GNU toolchain.

■ **Note** If the specified program is not found, GCC simply echoes the name of the specified program.

`-print-search-dirs`: This debugging option causes GCC to print the name of its installation directory and the program and library directories that it will search for mandatory files, and then exit. This option is useful when debugging installation problems reported by GCC. To resolve installation problems, you can either rebuild GCC correctly, or symlink or move any problematic components into one of the directories specified in the output of this command. You can often temporarily hack around installation problems by setting the environment variable `GCC_EXEC_PREFIX` to the full pathname (with a trailing `/`) of the directory where missing components are actually installed.

`-Q`: This debugging option causes GCC to print the name of each function as it is compiled and print some general statistics about each pass of the compiler.

`-rdynamic`: This linker option tells GCC to pass the `-export-dynamic` flag to the ELF linker on targets that support it, which instructs the linker to add information for all symbols in the code to the dynamic symbol table, not just those that are used.

`-remap`: This preprocessor command-line option enables special code in the preprocessor that is designed to work around filename limitations in filesystems, such as the MS-DOS FAT filesystem that only permits short, silly filenames.

`-S`: This output option causes GCC to stop after generating the assembler code for any specified input files. The assembler file for a given source file has the same name as the source file but has an `.s` extension instead of the original extension of the input source file.

`-s`: This linker option causes the linker to remove all symbol table and relocation information from the final executable.

`-save-temps`: This debugging option causes GCC to preserve all temporary files produced during the compilation process, storing them in the working directory of the compilation process. This produces `.i` (preprocessed C input) and `.s` (assembler) files for each file specified for compilation. These files have the same basename as the original input files but a different extension.

`-shared`: This linker option causes the linker to produce a shared object that can then be linked with other objects to form an executable. When using this option you should make sure that all of the shared objects that you will eventually link together are compiled with the same set of `-fpic`, `-fPIC`, or model suboption compiler options.

`-shared-libgcc` | `-static-libgcc`: These linker options cause the linker to force the use of the shared or static version of libgcc.a on systems that provide both. These options have no effect if a shared version of libgcc.a is not available. The shared version is generally preferable because this makes it easier to do operations such as throwing and catching exceptions across different shared libraries.

`-specs=`*`file`*: This directory search option tells GCC to process *`file`* to potentially override the default specs used by the compiler driver to identify default options and programs used during the compilation chain.

`-static`: This linker option causes the linker to prevent linking against shared libraries on systems that support them.

`-std=std`: This C language option tells GCC which C standard the input file is expected to conform to. You can use the features of a newer C standard even without specifying this option as long as they do not conflict with the default ISO C89 standard. Specifying a newer version of ISO C using `-std` essentially enables GCC's support for features in the specified standard to the default features found in ISO C89. Specifying a newer C standard changes the warnings that will be produced by the `-pedantic` option, which will display the warnings associated with the new base standard, even when you specify a GNU extended standard.

Possible values for `std` are the following:

- `c89` | `iso9899:1990`: ISO C89 (the same as using the `-ansi` switch).
- `iso9899:199409`: ISO C89 as modified in amendment 1.
- `c99` | `iso9899:1999`: ISO C99. This standard is not yet completely supported. For additional information, see `http://gcc.gnu.org/version/c99status.html`, where *version* is a major GCC version such as gcc-3.1, gcc-3.2, gcc-3.3, and so on. At the time this book was written the standard names `c9x` and `iso9899:199x` could also be specified but were deprecated.
- `gnu89`: ISO C89 with some GNU extensions and ISO C99 features. This is the default C standard used by gcc (i.e., the value used when the `-std` option is not given).
- `gnu99`: ISO C99 with some GNU extensions. This will become the default once ISO C99 is fully supported in GCC. The name `gnu9x` can also be specified, but is deprecated.

`-symbolic`: This linker option causes the linker to bind references to global symbols when building a shared object and to warn about any unresolved references.

`--sysroot=dir`: This directory search option tells GCC to use `dir` as the root of the file system in which header files and libraries are located.

`--target-help`: This C language output option causes GCC to print a list of options that are specific to the compilation target. Using this option is the easiest way to get an up-to-date list of all platform-specific GCC options for the target platform.

`-time`: This debugging option causes GCC to display summary information that lists user and system CPU time (in seconds) consumed by each step in the compilation process (cc1, as, ld, and so on). *User time* is the time actually spent executing the specified phase of the compilation process. *System time* is the time spent executing operating system routines on behalf of that phase of the compilation process.

`-traditional`: This C language option tells GCC to attempt to support some aspects of traditional C compilers and non-ANSI C code, and also automatically invokes the parallel `-traditional-cpp` option for GCC's internal C preprocessor. This option can only be used if the application being compiled does not reference header (#include) files that do not contain ISO C constructs. Though much beloved by K&R C fans, this option is deprecated and may disappear in a future release of GCC.

■ **Tip** When using this option, you may also want to specify the `-fno-builtin` option if your application implements functions with the same names as built-in GCC functions.

Some of the backward-compatibility features activated by this option are the following:

- Older function call sequence is acceptable; parameter types can be defined outside the parentheses that delimit the parameters but before the initial bracket for the function body.
- All automatic variables not declared using the `register` keyword are preserved by the `longjmp` function. In ISO C, any automatic variables not declared as volatile have undefined values after a return.
- All `extern` declarations are global even if they occur inside a function definition. This includes implicit function declarations.
- Newer keywords such as `typeof`, `inline`, `signed`, `const`, and `volatile` are not recognized. Alternative keywords such as `__typeof__`, `__inline__`, and so on, can still be used.
- Comparisons between pointers and integers are always allowed.
- Integer types `unsigned short` and `unsigned char` are always promoted to `unsigned int`.
- Floating-point literals that are out of range for that datatype are not an error.
- String constants are stored in writable space and are therefore not necessarily constant.
- The character escape sequences `\x` and `\a` evaluate as the literal characters `x` and `a`, rather than being a prefix for the hexadecimal representation of a character and a bell, respectively.

`-traditional-cpp`: This option when compiling C applications causes GCC to modify the behavior of its internal preprocessor to make it more similar to the behavior of traditional C preprocessors. See the GNU CPP manual for details (`http://gcc.gnu.org/onlinedocs/cpp`).

`-trigraphs`: This option while compiling a C application causes GCC to support ISO C trigraphs. The character set used in C source code is the 7 bit ASCII character set, which is a superset of a superset of the ISO 646-1983 Invariant Code Set. Trigraphs are sequences of three characters (introduced by two question marks) that the compiler replaces with their corresponding punctuation characters. Specifying the `-trigraphs` option enables C source files to be written using only characters in the ISO Invariant Code Set by providing an ISO-compliant way of representing punctuation or international characters for which there is no convenient graphical representation on the development system. The `-ansi` option implies the `-trigraphs` option because it enforces strict conformance to the ISO C standard.

The nine standard trigraphs and their replacements are the following:

```
Trigraph:       ??(  ??)      ??<   ??>        ??=      ??/   ??'  ??!     ??-
Replacement:    [     ]       {       }        #        \      ^     |      ~
```

`-U`*`NAME`*: This preprocessor option causes cpp to cancel any previous definition of `NAME`, regardless of whether it is built in or explicitly defined using the `-D` option.

`-undef`: This preprocessor option causes cpp not to predefine any system-specific macros. Standard system-independent macros are still defined.

`-V` *`VERSION`*: This target-related option causes GCC to attempt to run the specified version of GCC if multiple versions are installed on your system. Using this option can be handy if you have multiple versions of GCC installed on your system.

`-v`: This output option causes GCC to print the commands executed during each stage of compilation, along with the version number of each command.

`--version`: This output option tells GCC to display version and build information, and then exit.

`-W`: This diagnostic/warning option is a deprecated synonym for `-Wextra` in GCC 4.x.

`-Wa,option`: This causes GCC to pass `option` as an option to the assembler. If `option` contains commas, each comma is interpreted as a separator for multiple options.

`-Wabi`: This option when compiling C++ applications causes GCC to display a warning when it generates code that may not be compatible with the generic C++ ABI. Eliminating warnings of this type typically requires that you modify your code. The most common causes of these warnings are padding related, either when using bit fields or when making assumptions about the length of words or data structures.

`-Waggregate-return`: This diagnostic/warning option elicits a warning if any functions that return structures or unions are defined or called. (It also elicits a warning in languages where you can return an array.) This option is not implied by specifying the `-Wall` option.

`-Wall`: This diagnostic/warning option activates the majority of GCC's warnings.

■ **Note** Though you would assume that using GCC's `-Wall` option turns on all warnings, that is not the case. The following warning-related options are not automatically activated when you specify `-Wall` and must be separately specified if desired: `-W`, `-Waggregate-return`, `-Wbad-function-cast`, `-Wcast-align`, `-Wcast-qual`, `-Wconversion`, `-Wdisabled-optimization`, `-Werror`, `-Wfloat-equal`, `-Wformat-nonliteral`, `-Wformat-security`, `-Wformat=2`, `-Winline`, `-Wlarger-than-len`, `-Wlong-long`, `-Wmissing-declarations`, `-Wmissing-format-attribute`, `-Wmissing-noreturn`, `-Wmissing-prototypes`, `-Wnested-externs`, `-Wno-deprecated-declarations`, `-Wno-format-y2k`, `-Wno-format-extra-args`, `-Wpacked`, `-Wpadded`, `-Wpointer-arith`, `-Wredundant-decls`, `-Wshadow`, `-Wsign-compare`, `-Wstrict-prototypes`, `-Wtraditional`, `-Wundef`, and `-Wunreachable-code`. See the explanation of these options in this appendix for details on exactly what additional warnings they will generate.

`-Walways-true`: This warning option tells GCC to generate warnings about tests that are always true due to inherent attributes of the associated datatypes. This warning is enabled by `-Wall`.

`-Wassign-intercept`: This Objective C and Objective C++ warning option tells GCC to issue a warning whenever an Objective C assignment is intercepted by the garbage collector.

`-Wbad-function-cast`: This diagnostic/warning option when compiling a C application causes GCC to display a warning message when a function call is cast to a type that does not match the function declaration. Though this is something that many C programmers traditionally do, using this option can be useful to help you detect casts to datatypes of different sizes.

`-Wc++-compat`: This option tells GCC to issue a warning about any ISO C constructs used that are outside the common subset shared by ISO C and ISO C++.

`-Wcast-align`: This diagnostic/warning option causes GCC to display a warning message whenever a pointer is cast such that the target will need to change the alignment of the specified data structure. For example, casting a `char *` to an `int *` on machines where integers can only be accessed at 2- or 4-byte boundaries would generate this warning.

`-Wcast-qual`: This diagnostic/warning option causes GCC to display a warning message whenever casting a pointer removes a type qualifier from the target type. For example, casting a `const char *` to an ordinary `char *` would generate this warning.

`-Wchar-subscripts`: This diagnostic/warning option causes GCC to display a warning message whenever an array subscript has type `char`.

`-Wcomment`: This diagnostic/warning option causes GCC to display a warning message whenever a comment-start sequence (`/*`) appears within another (`/*`) comment or whenever a newline is escaped within a `//` comment. This is an incredibly helpful option to detect most cases of "comment overflow."

`-Wconversion`: This diagnostic/warning option causes GCC to display a warning message whenever a prototype causes a type conversion that is different from what would happen to the same argument in the absence of a prototype. This helps identify conversions that would change the width or signedness of a variable, which is a common cause of error on machines with alignment requirements. This option will not generate warnings for explicit casts such as `(unsigned) -1` because these will be preresolved during compilation.

`-Wctor-dtor-privacy`: This diagnostic/warning option when compiling a C++ application causes GCC to display a warning whenever a class seems unusable, because all the constructors or destructors are private and the class has no friends (i.e., it grants no access to other classes or functions) or public static member functions.

`-Wdeclaration-after-statement`: This C language warning option tells GCC to issue a warning whenever a declaration follows a statement in a basic block. This is not supported by ISO C90 or versions of GCC prior to 3.0, but was introduced in C99 and is now allowed by GCC.

`-Wdisabled-optimization`: This diagnostic/warning option causes GCC to display a warning message whenever a requested optimization pass is disabled or skipped. This rarely indicates a problem with your GCC installation or code but instead usually means that GCC's optimizers are simply unable to handle the code because of its size, complexity, or the amount of time that the requested optimization pass would require.

`-Wdiv-by-zero`: This diagnostic/warning option causes GCC to display a warning message whenever the compiler detects and attempts to divide an integer by zero. You can disable this warning by using the `-Wno-div-by-zero` option. This warning is not generated when an application attempts floating-point division by zero, because this can occasionally be used in applications to generate values for infinity and NaN.

`-Weffc++`: This diagnostic/warning option, when compiling a C++ application, causes GCC to display a warning message whenever application code violates various guidelines described in Scott Meyers' *Effective C++* (Addison-Wesley, 2005. ISBN: 0-321-33487-6) and *More Effective C++* (Addison-Wesley, 1995. ISBN: 0-201-63371-X), such as the following:

- Define a copy constructor and an assignment operator for classes with dynamically allocated memory (Item 11, *Effective C++*).
- Prefer initialization to assignment in constructors (Item 12, *Effective C++*).
- Make destructors virtual in base classes (Item 14, *Effective C++*).
- Have `operator=` return a reference to `*this` (Item 15, *Effective C++*).
- Do not try to return a reference when you must return an object (Item 23, *Effective C++*).
- Distinguish between prefix and postfix forms of increment and decrement operators (Item 6, *More Effective C++*).
- Never overload the operators &&, ||, or , (Item 7, *More Effective C++*).

■ **Note** Ironically, some of the standard header files used by GCC do not follow these guidelines, so activating this warning option may generate unexpected messages about system files. You can ignore warnings from outside your code base or use a utility such as `grep -v` to filter out those warnings.

`-Werror`: This diagnostic/warning option causes GCC to make all warnings into errors.

`-Werror-implicit-function-declaration`: This diagnostic/warning option causes GCC to display a warning message whenever a function is used before it has been declared.

`-Wextra`: This diagnostic/warning option causes GCC to display extra warning messages when it detects any of the following events in the code that is being compiled:

- A function can return either with or without a value. If a function returns a value in one case, both a return statement with no value (such as `return;`) or an implicit return after reaching the end of a function will trigger a warning.
- The left side of a comma expression has no side effects. (A *comma expression* is an expression that contains two operands separated by a comma. Although GCC evaluates both operands, the value of the expression is the value of the right operand. The left operand of a comma expression is used to do an assignment or produce other side effects—if it produces a value, it is discarded by GCC.) To suppress the warning, cast the left-side expression to `void`.
- An unsigned value is compared against zero using `<` or `<=`.
- A comparison such as `x <= y <= z` appears. GCC interprets this as `((x <= y) < z)`, which compares the return value of the comparison of *x* and *y* against the value of *z*, which is usually not what is intended.
- Storage-class specifiers such as `static` are not the first things in a declaration, which is suggested by modern C standards.
- A return type of a function has a type qualifier. This has no effect because the return value of a function is not an assigned lvalue.
- Unused arguments to a function call are present. This warning will only be displayed if the `-Wall` or `-Wunused` options are also specified.
- A comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned. To eliminate this warning, specify the `-Wno-sign-compare` command-line option.
- An aggregate has a partly bracketed initializer, which is usually seen when initializing data structures that contain other data structures. The values passed to an internal data structure must also be enclosed within brackets.
- An aggregate has an initializer that does not initialize all members of the structure.

`-Wfatal-errors`: This warning option tells GCC to abort compilation whenever any error occurs, rather than trying to continue.

`-Wfloat-equal`: This diagnostic/warning option causes GCC to display a warning message whenever floating-point values are compared for equality. Because floating-point values are often used as approximations for infinitely precise real numbers, it is not always possible to precisely compare such approximation. If you are doing this, a better suggestion is to compare floating-point values by determining whether they fall within an acceptable range of values by using relational operators.

`-Wformat`: This diagnostic/warning option causes GCC to display a warning message whenever the arguments to calls to `printf`, `scanf`, `strftime` (X11), `strfmon` (X11), and similar functions do not have types appropriate to the specified format string. If the `-pedantic` option is used with this option, warnings will be generated for any use of format strings that are not consistent with the programming language standard being used.

`-Wformat=2`: This diagnostic/warning option is the same as explicitly invoking the `-Wformat`, `-Wformat-nonliteral`, and `-Wformat-security` options.

`-Wformat-nonliteral`: This diagnostic/warning option causes GCC to display a warning message whenever a format string is not a string literal and therefore cannot be checked, unless the format function takes its format arguments as a variable argument list (`va_list`).

`-Wformat-security`: This diagnostic/warning option causes GCC to display a warning message whenever calls to the `printf()` and `scanf()` functions use a format string that is not a string literal and there are no format arguments, as in `printf (foo);`. At the time this book was written, this option was a subset of the warnings generated by the `-Wformat-nonliteral` option but was provided to explicitly detect format strings that may be security holes.

`-Wformat-y2k`: This warning option when `-Wformat` is also specified tells GCC to also issue warnings about `strftime()` formats that may produce a two-digit year.

`-Wimplicit`: This diagnostic/warning option is the same as explicitly invoking the `-Wimplicit-int` and `-Wimplicit-function-declaration` options.

`-Wimplicit-function-declaration`: This diagnostic/warning option causes GCC to display a warning message whenever a function is used before being declared.

`-Wimplicit-int`: This diagnostic/warning option causes GCC to display a warning message whenever a declaration does not specify a type, which therefore causes the declared function or variable to default to being an integer.

`-Wimport`: This warning option tells GCC to issue a warning the first time that a #import directive is used.

`-Winit-self`: This C, C++, and Objective C warning option, when the `-Wuninitialized` option and optimization levels 1 and higher are being used, tells GCC to issue warnings about any uninitialized variables that are initialized by being set to themselves.

`-Winline`: This diagnostic/warning option causes GCC to display a warning message whenever a function that was declared as inline cannot be inlined.

`-Winvalid-pch`: This warning option tells GCC to issue a warning if a precompiled header is found in the search path but cannot be used.

`-Wl,option`: This causes GCC to pass `option` as an option to the linker. If `option` contains commas, each is interpreted as a separator for multiple options.

`-Wlarger-than-len`: This diagnostic/warning option causes GCC to display a warning message whenever an object of larger than `len` bytes is defined.

`-Wlong-long`: This diagnostic/warning option causes GCC to display a warning message whenever the `long long` type is used. This warning option is automatically enabled when the `-pedantic` option is specified. You can inhibit these warning messages in this case by using the `-Wno-long-long` option.

`-Wmain`: This diagnostic/warning option causes GCC to display a warning message whenever the type of `main()` or the number of arguments passed to it is suspicious. A program's main routine should always be an externally linked function that returns an integer value and takes either zero, two, or three arguments of the appropriate types.

`-Wmissing-braces`: This diagnostic/warning option causes GCC to display a warning message whenever an aggregate or union initializer is not correctly bracketed so that it explicitly follows the conventions of the aggregate or union. As an example, the following expression would generate this warning:

```
int a[2][2] = { 0, 1, 2, 3 };
```

`-Wmissing-declarations`: This diagnostic/warning option causes GCC to display a warning message whenever a global function is defined without a previous declaration, even if the definition itself provides a prototype. Using this option detects global functions that are not declared in header files.

`-Wmissing-field-initializers`: This warning option tells GCC to issue a warning if a structure initializer does not initialize all of the fields in the structure.

`-Wmissing-format-attribute`: This diagnostic/warning option for C programs causes GCC to display a warning message whenever a function such as `printf()` or `scanf()` contains a format string that contains more attributes than are provided in subsequent arguments to the call. If the `-Wformat` option is also specified, GCC will also generate warnings about similar occurrences in other functions that appear to take format strings.

`-Wmissing-include-dirs`: This C, C++, and Objective C warning option tells GCC to issue a warning if a user-specified include directory does not exist.

`-Wmissing-noreturn`: This diagnostic/warning option causes GCC to display a warning message whenever functions are used that might be candidates for the `noreturn` attribute `(__attribute__((noreturn)) prototype;)`.

`-Wmissing-prototypes`: This diagnostic/warning option causes GCC to display a warning message whenever compiling a C application in which a global function is defined without a previous prototype declaration and is intended to detect global functions that are not declared in header files. This warning is issued even if the definition itself provides a prototype.

`-Wmultichar`: This diagnostic/warning option causes GCC to display a warning message whenever a multicharacter constant (e.g., foo) is used. This option is enabled by default, but can be disabled by specifying the `-Wno-multichar` option. Multicharacter constants should not be used in portable code because their internal representation is platform-specific.

`-Wnested-externs`: This diagnostic/warning option when compiling a C application causes GCC to display a warning message whenever an `extern` declaration is encountered within a function.

`-Wno-deprecated`: This diagnostic/warning option causes GCC not to display a warning message whenever deprecated features are used.

`-Wno-deprecated-declarations`: This diagnostic/warning option causes GCC to not display a warning message whenever functions, variables, and types marked as deprecated (through the `deprecated` attribute) are used.

`-Wno-div-by-zero`: This warning option tells GCC to suppress compile-time warnings about integer division by zero.

`-Wno-endif-labels`: This warning option tells GCC to suppress warnings whenever `#else` or `#endif` statements are followed by additional text.

`-Wno-format-y2k`: This diagnostic/warning option causes GCC not to display a warning message whenever `strftime()` formats are used that may yield only a two-digit year.

`-Wno-format-extra-args`: This diagnostic/warning option when `-Wformat` is also specified causes GCC to not display a warning message whenever excess arguments are supplied to a `printf()` or `scanf()` function. Extra arguments are ignored, as specified in the C standard. Warnings will still be displayed if the unused arguments are not all pointers and lie between used arguments that are specified with $ operand number specifications.

`-Wno-import`: This diagnostic/warning option causes GCC to not display a warning message whenever `#import` statements are encountered in an Objective C application. (The `#import` statement is identical to C's `#include` statement, but will not include the same include file multiple times.)

`-Wno-int-to-pointer-cast`: This C language warning option tells GCC to suppress warnings when integers of one size are cast to pointers of another.

`-Wno-invalid-offsetof`: This C++ warning option tells GCC to suppress warnings from applying the `offsetof()` macro to a non-POD (plain old data) type, which is undefined according to the 1998 ISO C++ standard.

`-Wno-multichar`: This warning option tells GCC to suppress warnings if multicharacter constants are used.

`-Wno-non-template-friend`: This diagnostic/warning option when compiling a C++ application causes GCC not to display a warning message whenever nontemplatized friend functions are declared within a template. The C++ language specification requires that friends with unqualified IDs declare or define an ordinary, nontemplate function. Because unqualified IDs could be interpreted as a particular specialization of a templatized function in earlier versions of GCC, GCC now checks for instances of this in C++ code by using the `-Wnon-template-friend` option as a default. The `-Wno-non-template-friend` option can be used to disable this check but keep the conformant compiler code.

`-Wno-pmf-conversions`: This diagnostic/warning option causes GCC not to display a warning message whenever C++ disables the diagnostic for converting a bound pointer of a member function to a plain pointer.

`-Wno-pointer-to-int-cast`: This C language warning option tells GCC to suppress warnings whenever an integer of one size is cast to a pointer type of another size.

`-Wno-pragmas`: This warning option tells GCC to suppress warnings related to misuse of pragmas, including invalid syntax, incorrect parameters, or conflicts with other pragmas.

`-Wno-protocol`: This diagnostic/warning option when compiling an Objective C application causes GCC to not display a warning message if methods required by a protocol are not implemented in the class that adopts it.

`-Wno-return-type`: This diagnostic/warning option causes GCC to suppress warning messages whenever a function is defined with a return type that defaults to `int`, or when a return without a value is encountered in a function whose return type is not `void`. This option does not suppress warning messages when compiling C++ applications that contain nonsystem, nonmain functions without a return type.

`-Wno-sign-compare`: This diagnostic/warning option causes GCC to suppress displaying a warning message whenever a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned.

`-Wnon-template-friend`: This option causes GCC to check for unqualified IDs that could be interpreted as a particular specialization of a templatized function when compiling C++ applications under earlier versions of GCC and display an error message. The C++ language specification requires that friends with unqualified IDs declare or define an ordinary, nontemplate function. This option is enabled by default.

`-Wnon-virtual-dtor`: This diagnostic/warning option when compiling a C++ application causes GCC to not display a warning message whenever a class declares a nonvirtual destructor that should probably be virtual because the class may be used polymorphically.

`-Wnonnull`: This option tells GCC to issue a warning whenever a null pointer is passed as an argument that is marked as requiring a nonnull value by the nonnull attribute.

`-Wold-style-cast`: This diagnostic/warning option when compiling a C++ application causes GCC to display a warning message whenever a `C-stylestyle` cast to a non-`void` type is used within a C++ program. Newer cast statements such as `const_cast`, `reinterpret_cast`, and `static_cast` should be used instead, because they are less vulnerable to unintended side effects.

`-Wold-style-definition`: This C language warning option tells GCC to issue a warning whenever an old-style function definition is encountered.

`-Woverlength-strings`: This warning option tells GCC to issue a warning about strings that are longer than the specified maximum in a given C standard. In ISO C89 the limit is 509 characters. In ISO C99 the limit is 4,095 characters. This option is enabled by the `-pedantic` option.

`-Woverloaded-virtual`: This diagnostic/warning option when compiling a C++ application causes GCC to display a warning message whenever a function declaration hides virtual functions from a base class, typically because a virtual function with the same name is already present in a base class.

`-Wp,option`: This causes GCC to pass `option` as an option to the preprocessor. If `option` contains commas, each comma is interpreted as a separator for multiple options.

`-Wpacked`: This diagnostic/warning option causes GCC to display a warning message whenever a structure is specified as *packed*, but the `packed` attribute has no effect on the layout or size of the structure.

`-Wpadded`: This diagnostic/warning option causes GCC to display a warning message whenever padding is included in a data structure, regardless of whether it is used to align a single element or the entire data structure. This warning is displayed because it is often possible to reduce the size of the structure simply by rearranging its components.

`-Wparentheses`: This diagnostic/warning option causes GCC to display a warning message whenever parentheses are omitted in certain contexts, such as when there is an assignment in a context where a truth value is expected, when operators are nested whose precedence is commonly confused, or when there may be confusion about the `if` statement to which an `else` branch belongs. Warnings caused by the latter two problems can easily be corrected by adding brackets to explicitly identify nesting.

`-Wpointer-arith`: This diagnostic/warning option causes GCC to display a warning message whenever anything depends on the size of a function type or of `void`. GNU C assigns these types a size of 1 for convenience in calculations and pointer comparisons.

`-Wpointer-sign`: This C and Objective C language warning option tells GCC to issue a warning when assigning or passing pointers as arguments where source and destination have different signedness. This option is enabled by the `-Wall` or `-pedantic` options.

`-Wredundant-decls`: This diagnostic/warning option causes GCC to display a warning message whenever anything is declared more than once in the same scope.

`-Wreorder`: This diagnostic/warning option when compiling a C++ application causes GCC to display a warning message whenever the order of member initializers given in the code does not match the order in which they were declared. A warning is displayed to inform you that GCC is reordering member initializers to match the declaration.

`-Wreturn-type`: This diagnostic/warning option causes GCC to display a warning message whenever a function is defined with a return type that defaults to `int`, or when a return without a value is encountered in a function whose return type is not `void`. When compiling C++ applications, nonsystem functions without a return type (and which are not `main()`) always produce this error message, even when the `-Wno-return-type` option is encountered. This option is active by default.

`-Wselector`: This diagnostic/warning option when compiling an Objective C application causes GCC to display a warning message whenever a selector defines multiple methods of different types.

`-Wsequence-point`: This diagnostic/warning option when compiling a C application causes GCC to display a warning message whenever the compiler detects code that may have undefined semantics because of violations of sequence-point rules in the C standard. Sequence-point rules help the compiler order the execution of different parts of the program and can be violated by code sequences with undefined behavior such as `a = a++`, `a[n] = b[n++]`, and `a[i++] = i;`. Some more complicated cases may not be identified by this option, which may also occasionally give a false positive.

`-Wshadow`: This diagnostic/warning option causes GCC to display a warning message whenever a local variable shadows another local variable, parameter, or global variable, or whenever a built-in function is shadowed.

`-Wsign-compare`: This diagnostic/warning option causes GCC to display a warning message whenever a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned. This option is automatically invoked when you specify the `-W` option.

`-Wsign-promo`: This diagnostic/warning option when compiling a C++ application causes GCC to display a warning message whenever overload resolution chooses a promotion from an unsigned or enumerated type to a signed type over a conversion to an unsigned type of the same size. Earlier versions of GCC would try to preserve unsignedness.

`-Wstack-protector`: This option when `-fstack-protect` is also specified tells GCC to issue a warning about any functions that will not be protected.

`-Wstrict-aliasing`: This option tells GCC to issue warnings for any code that might break the strict aliasing rules that the compiler is using as a basis for optimization. This option is enabled by the `-Wall` option.

`-Wstrict-aliasing=2`: This warning option when `-fstrict-warning` is also specified tells GCC to perform stricter checks for aliasing violations than the `-Wstrict-aliasing` option. This produces more warnings but may generate warnings about some cases that are actually safe.

`-Wstrict-null-sentinel`: This C++ language option tells GCC to issue a warning whenever an uncast `NULL` is being used as a sentinel, which is not portable across different C++ compilers.

`-Wstrict-prototypes`: This diagnostic/warning option when compiling a C application causes GCC to display a warning message whenever a function is declared or defined without specifying the types of its arguments.

`-Wswitch`: This diagnostic/warning option is the same as specifying both the `–Wswitch-default` and the `–Wswitch-enum` warning options.

`-Wswitch-default`: This warning option tells GCC to issue a warning whenever a switch statement does not have a default case.

`-Wswitch-enum`: This warning option tells GCC to issue a warning whenever a switch statement has an index of an enumerated type but lacks a case statement for one or more possible values of that type.

`-Wsynth`: This diagnostic/warning option when compiling a C++ application causes GCC to display a warning message whenever GCC's synthesis behavior does not match that of Cfront. Cfront is a family of C++ compilers from many different vendors that serves as a front end for C++ code, translating it so that it can subsequently be compiled by a standard C compiler.

`-Wsystem-headers`: This diagnostic/warning option causes GCC to display a warning message whenever potentially invalid constructs are found in system header files. Using this command-line option tells GCC to display warnings about system headers as if they occurred in application code. To display warnings about unknown pragmas found in system headers, you must also specify the `-Wunknown-pragmas` option.

`-Wtraditional`: This diagnostic/warning option when compiling a C application causes GCC to display a warning message whenever constructs are encountered that behave differently in traditional and ISO C or are only found in ISO C, and for problematic constructs that should generally be avoided. Some examples of these are the following:

- Macro parameters that appear within string literals in the macro body. Traditional C supports macro replacement within string literals, but ISO C does not.
- Preprocessor directives that do not begin with the hash symbol (`#`) as the first character on a line. Preprocessor directives such as `#pragma` that are not supported by traditional C can thus be "hidden" by indenting them. For true portability, you may want to generally avoid preprocessor directives such as `#elif` that are not supported by traditional C.
- Functionlike macros that appear without arguments.
- The `U` integer constant suffix, or the `F` or `L` floating-point constant suffixes.
- Functions that are declared external in one block and subsequently used after the end of the block.
- A switch statement that has an operand of type `long`.
- Nonstatic function declarations that follow static ones.
- Integer constants. The ISO type of a decimal integer constant has a different width or signedness from its traditional type.
- ISO string concatenation.
- Initialization of automatic aggregates, which are nonstatic local arrays and structures.
- Conflicts between identifiers and labels.
- Union initialization of nonzero unions.
- Prototype conversions between fixed and floating-point values and vice versa. You can use the `-Wconversion` option to display additional warnings related to possible conversion problems.

`-Wtrigraphs`: This diagnostic/warning option causes GCC to display a warning message whenever trigraphs without comments are encountered that might change the meaning of the program.

`-Wundeclared-selector`: This Objective C and Objective C++ language option tells GCC to issue a warning if an `@selector()` expression referring to an undeclared selector is encountered. This option checks for this condition whenever an `@selector()` expression is encountered, unlike the similar `-Wselector` option, which checks for this condition in the final stages of compilation.

`-Wundef`: This diagnostic/warning option causes GCC to display a warning message whenever an undefined identifier is evaluated in a `#if` preprocessor directive.

`-Wuninitialized`: This diagnostic/warning option causes GCC to display a warning message whenever an automatic variable is used without first being initialized, or if an existing nonvolatile variable may be changed by a `setjmp` call. These warnings are only generated when using the `-O`, `-O1`, `-O2`, or `-O3` optimization options, and then only for nonvolatile variables that are candidates for register allocation.

`-Wunknown-pragmas`: This diagnostic/warning option causes GCC to display a warning message whenever it encounters an unknown `#pragma` preprocessor directive.

`-Wunreachable-code`: This diagnostic/warning option causes GCC to display a warning message whenever GCC detects code that will never be executed.

`-Wunsafe-loop-optimizations`: This warning option tells GCC to issue a warning when a loop cannot be optimized because the compiler cannot make assumptions about the bounds of loop indices.

`-Wunused`: This diagnostic/warning option provides a convenient shortcut for specifying all of the `-Wunused-function`, `-Wunused-label`, `-Wunused-value`, and `-Wunused-variable` options. In order to get a warning about an unused function parameter, you must either specify the `-W` and `-Wunused` options or separately specify the `-Wunused-parameter` option.

`-Wunused-function`: This diagnostic/warning option causes GCC to display a warning message whenever a static function is declared but not defined, or when a noninline static function is not used.

`-Wunused-label`: This diagnostic/warning option causes GCC to display a warning message whenever a label is declared but not used.

`-Wunused-parameter`: This diagnostic/warning option causes GCC to display a warning message whenever a function parameter is unused aside from its declaration.

`-Wunused-value`: This diagnostic/warning option causes GCC to display a warning message whenever a statement computes a result that is not used.

`-Wunused-variable`: This diagnostic/warning option causes GCC to display a warning message whenever a local variable or nonconstant static variable is unused aside from its declaration.

`-Wvariadic-macros`: This option tells GCC to issue a warning if variadic macros are used in pedantic ISO C90 mode, or if alternate GNU syntax for these is used in pedantic C99 mode. This option is active by default.

`-Wvolatile-register-var`: This warning option tells GCC to issue a warning if a register variable is declared as volatile.

`-Wwrite-strings`: This C and C++ language warning option tells GCC to impose a type of `const char[LENGTH]` on string constants and to issue a warning when copying the address of one into a non-`const char` pointer. This option is enabled by default for C++ programs.

`-w`: This diagnostic/warning option causes GCC not to display any warning messages.

`-Xassembler` *`option`*: This causes GCC to pass `option` as an option to the assembler, and is often used to supply system-specific assembler options that GCC does not recognize. Each option is a single token. If you need to pass an option that takes an argument, you must use the `–Xassembler` option twice, once for the option and once for the argument.

`-Xlinker` *`option`*: This causes GCC to pass `option` as an option to the linker, and is often used to supply system-specific linker options that GCC does not recognize. Each option is a single token. If you need to pass an option that takes an argument, you must use the `–Xlinker` option twice, once for the option and once for the argument.

`-Xpreprocessor` *`option`*: This causes GCC to pass `option` as an option to the preprocessor, and is often used to supply system-specific preprocessor options that GCC does not recognize. Each option is a single token. If you need to pass an option that takes an argument, you must use the `–Xpreprocessor` option twice, once for the option and once for the argument.

`-x [`*`language`*`|none]`: This output option identifies the output *language* to be generated rather than letting the compiler choose a default based on the extension of the input file. This option applies to all following input files until the next `-x` option. Possible values for *language* are `ada`, `assembler`, `assembler-with-cpp`, `c`, `c-header`, `c++`, `c++-cpp-output`, `cpp-output`, `f77`, `f77-cpp-output`, `java`, `objc-cpp-output`, `objective-c`, and `ratfor`. Specifying *none* turns off the language specification, reverting to GCC's defaults based on the extension of the input file.

APPENDIX B

Machine- and Processor-Specific Options for GCC

GCC provides hundreds of machine-specific options that you will rarely need to use unless you are compiling for a specific platform and need to take advantage of some of its unique characteristics. This appendix provides a summary and a discussion of machine-specific options for GCC, organized by the platform to which they are relevant. For your convenience if you are using versions of GCC other than 4.x, this appendix includes machine- and processor-specific options for some older targets that are no longer supported in the GCC 4.x compiler family. Where obsolete platform options are discussed, a note identifies the GCC versions to which they are relevant.

Machine- and architecture-specific configuration information for GCC is stored in the `gcc/config` subdirectory of a GCC source code installation. In theory, each supported system has its own directory that contains general configuration information as well as specific information for supported variants of that processor or architecture. The one exception to this rule is Darwin support, which lives in the `gcc/config` subdirectory because it is an OS kernel and execution environment rather than an architecture (and actually spans multiple architectures). It is treated as a machine in this appendix because there are a large number of Darwin-specific GCC options available, thanks to the fact that it is open source, it runs on both x86 and PPC architectures, and Apple depends on it for the lovely Mac OS X.

The majority of the machine- and CPU-specific options available in GCC are specific values for GCC's `-m` command-line option, which enables you to identify characteristics of the machine for which GCC is generating code.

Note The options discussed in this section are only relevant if you are using a version of GCC that was either compiled to run directly on the specified platform (not always possible) or if you are using a version of GCC that has been built as a cross-compiler, generating binaries for the specified platform even though it is actually executing on another platform. For more information about cross-compilers, see Chapter 14.

Alpha Options

The 64-bit Alpha processor family was originally developed by Digital Equipment Corporation (DEC) and inherited by its purchasers, Compaq Computer and (later) Hewlett-Packard. The Alpha was an extremely fast processor for its time whose widespread adoption was hampered by VMS and DEC's series of one-night stands with a variety of flavors of Unix.

GCC options available when compiling code for Unix-like operating systems running on the DEC Alpha family of processors are the following:

`-malpha-as`: Specifying this option tells GCC to generate code that is to be assembled by the DEC assembler.

`-mbuild-constants`: Specifying this option causes GCC to construct all integer constants using code that checks to see if the program can construct the constant from smaller constants in two or three instructions. If it cannot, GCC outputs the constant as a literal and generates code to load it from the data segment at runtime. Normally, GCC only performs this check on 32- or 64-bit integer constants. The goal of this option is to keep constants out of the data segment and in the code segment whenever possible. This option is typically used when building a dynamic loader for shared libraries, because the loader must be able to relocate itself before it can locate its data segment.

`-mbwx`: Specifying this option tells GCC to generate code to use the optional BWX instruction set. The default is to use the instruction sets supported by a CPU type specified by using the `-mcpu=CPU-type` option, or the instruction sets supported on the CPU on which GCC was built if the `-mcpu=CPU-type` option was not specified.

`-mcix`: Specifying this option tells GCC to generate code to use the optional CIX instruction set. The default is to use the instruction sets supported by a CPU type specified by using the `-mcpu=CPU-type` option, or the instruction sets supported on the CPU on which GCC was built if the `-mcpu=CPU-type` option was not specified.

`-mcpu=CPU-type`: Specifying this option tells GCC to use the instruction set and instruction scheduling parameters that are associated with the machine type `CPU-type`. Instruction scheduling is the phase of compilation that sequences instructions in order to maximize possible parallelism and minimize any time that instructions spend waiting for results of other instructions. You can specify either the chip name (EV-style name) or the corresponding chip number. If you do not specify a processor type, GCC will default to the processor on which the GCC was built. Supported values for `CPU-type` are the following:

- `ev4` | `ev45` | `21064`: Schedule as an EV4 without instruction set extensions
- `ev5` | `21164`: Schedule as an EV5 without instruction set extensions
- `ev56` | `2164a`: Schedule as an EV5 and support the BWX instruction set extension
- `pca56` | `21164pc` | `21164PC`: Schedule as an EV5 and support the BWX and MAX instruction set extensions
- `ev6` | `21264`: Schedule as an EV6 and support the BWX, FIX, and MAX instruction set extensions
- `ev67` | `21264a`: Schedule as an EV6 and support the BWX, CIX, FIX, and MAX instruction set extensions

`-mexplicit-relocs`: Specifying this option tells GCC to generate code that explicitly marks which type of symbol relocation should apply to which instructions. See the discussion of the `-msmall-data` and `–mlarge-data` options for additional information related to symbol relocation when the `-explicit-relocs` option is specified. This option is essentially a workaround for older assemblers that could only do relocation by using macros.

`-mfix`: Specifying this option tells GCC to generate code to use the optional FIX instruction set. The default is to use the instruction sets supported by a CPU type specified by using the `-mcpu=CPU-type` option, or the instruction sets supported on the CPU on which GCC was built if the `-mcpu=CPU-type` option was not specified.

`-mfloat-ieee`: Specifying this option tells GCC to generate code that uses IEEE single and double precision instead of VAX F and G floating-point arithmetic.

`-mfloat-vax`: Specifying this option tells GCC to generate code that uses VAX F and G floating-point arithmetic instead of IEEE single and double precision.

`-mfp-regs`: Specifying this option causes GCC to generate code that uses the floating-point register (FPR) set. This is the default.

`-mfp-rounding-mode=`*`rounding-mode`*: Using this option enables you to specify the IEEE rounding mode used to round off floating-point numbers. The *`rounding-mode`* must be one of the following:

- `c`: Chopped rounding mode. Floating-point numbers are rounded toward zero.
- `d`: Dynamic rounding mode. A field in the floating-point control register (FPCR) (see the reference manual for the Alpha architecture and processors) controls the rounding mode that is currently in effect. The C library initializes this register for rounding toward plus infinity. Unless your program modifies this register, a `rounding-mode` of `d` means that floating-point numbers are rounded toward plus infinity.
- `m`: Round toward minus infinity.
- `n`: Normal IEEE rounding mode. Floating-point numbers are rounded toward the nearest number that can be represented on the machine, or to the nearest even number that can be represented on the machine if there is no single nearest number.

`-mfp-trap-mode=`*`trap-mode`*: Specifying this option controls what floating-point related traps are enabled. *`trap-mode`* can be set to any of the following values:

- `n`: Normal. The only traps that are enabled are the ones that cannot be disabled in software, such as the trap for division by zero. This is the default setting.
- `su`: Safe underflow. This option is similar to `u trap-mode`, but marks instructions as safe for software completion. (See the Alpha architecture manual for details.)
- `sui`: Safe underflow inexact. This enables inexact traps as well as the traps enabled by `su trap-mode`.
- `u`: Underflow. This enables underflow traps as well as the traps enabled by the normal `trap-mode`.

`-mgas`: Specifying this option tells GCC to generate code that is to be assembled by the GNU assembler.

`-mieee`: Specifying this option causes GCC to generate fully IEEE-compliant code except that the `INEXACT-FLAG` is not maintained. The Alpha architecture provides floating-point hardware that is optimized for maximum performance and is generally compliant with the IEEE floating-point standard. This GCC option makes generated code fully compliant.

When using this option, the preprocessor macro `_IEEE_FP` is defined during compilation. The resulting code is less efficient but is able to correctly support denormalized numbers and exceptional IEEE values such as Not-a-Number and plus/minus infinity.

`-mieee-conformant`: Specifying this option tells GCC to mark the generated code as being IEEE conformant. You must not use this option unless you also specify `-mtrap-precision=i` and either `-mfp-trap-mode=su` or `-mfp-trap-mode=sui`. Specifying this option inserts the line `.eflag 48` in the function prologue of the generated assembly file. Under DEC Unix, this line tells GCC to link in the IEEE-conformant math library routines.

`-mieee-with-inexact`: Specifying this option tells GCC to generate fully IEEE-compliant code and to maintain the status of the `INEXACT-FLAG`. In addition to the preprocessor macro `_IEEE_FP`, the macro `_IEEE-FP-EXACT` is also defined during compilation. Because little code depends on the `EXACT-FLAG`, this option is rarely used. Code compiled with this option may execute more slowly than code generated by default.

`-mlarge-data`: Specifying this option causes GCC to store all data objects in the program's standard data segment. This increases the possible size of the data area to just below 2GB, but may require additional instructions to access data objects. Programs that require more than 2GB of data must use `malloc` or `mmap` to allocate the data in the heap instead of in the program's data segment.

■ **Note** When generating code for shared libraries on an Alpha, specifying the `-fPIC` option implies the `-mlarge-data` option.

`-mlarge-text`: Specifying this option causes GCC not to make any assumptions about the final size of the code that it produces, increasing overall code size but providing more flexibility. This option is the default.

`-mlong-double-64`: Specifying this option tells GCC to use the default size of 64 bits for both the `long double` and `double` datatypes. This is the default.

`-mlong-double-128`: Specifying this option tells GCC to double the size of the `long double` datatype to 128 bits. By default, it is 64 bits and is therefore internally equivalent to the `double` datatype.

`-mmax`: Specifying this option tells GCC to generate code to use the optional MAX instruction set. The default is to use the instruction sets supported by a CPU type specified by using the `-mcpu=CPU-type` option, or the instruction sets supported on the CPU on which GCC was built if the `-mcpu=CPU-type` option was not specified.

`-mmemory-latency=`*`time`*: Specifying this option tells GCC how to set the latency that the scheduler should assume for typical memory references as seen by the application. The specified value supplied for *time* depends on the memory access patterns used by the application and the size of the external cache on the machine. Supported values for *time* are the following:

- `number`: A decimal number representing clock cycles.
- `L1` | `L2` | `L3` | `main`: Use internal estimates of the number of clock cycles for typical EV4 and EV5 hardware for the Level 1, 2, and 3 caches, as well as `main` memory. (The level 1, 2, and 3 caches are also often referred to as Dcache, Scache, and Bcache, respectively.) Note that L3 is only valid for EV5.

`-mno-bwx`: Specifying this option tells GCC not to generate code to use the optional BWX instruction set. The default is to use the instruction sets supported by a CPU type specified by using the `-mcpu=CPU-type` option, or the instruction sets supported on the CPU on which GCC was built if the `-mcpu=CPU-type` option was not specified.

`-mno-cix`: Specifying this option tells GCC not to generate code to use the optional CIX instruction set. The default is to use the instruction sets supported by a CPU type specified by using the `-mcpu=CPU-type` option, or the instruction sets supported on the CPU on which GCC was built if the `-mcpu=CPU-type` option was not specified.

`-mno-explicit-relocs`: Specifying this option tells GCC to generate code that follows the old Alpha model of generating symbol relocations through assembler macros. Use of these macros does not allow optimal instruction scheduling.

`-mno-fix`: Specifying this option tells GCC not to generate code to use the optional FIX instruction set. The default is to use the instruction sets supported by a CPU type specified by using the `-mcpu=CPU-type` option, or the instruction sets supported on the CPU on which GCC was built if the `-mcpu=CPU-type` option was not specified.

`-mno-fp-regs`: Specifying this option causes GCC to generate code that does not use the floating-point register set. When the floating-point register set is not used, floating-point operands are passed in integer registers as if they were integers and floating-point results are passed in `'$0'` instead of `'$f0'`. This is a nonstandard calling sequence, so any function with a floating-point argument or return value called by code compiled with `-mno-fp-regs` must also be compiled with that option. Specifying this option implies the `-msoft-float` option.

`-mno-max`: Specifying this option tells GCC not to generate code to use the optional MAX instruction set. The default is to use the instruction sets supported by a CPU type specified by using the `-mcpu=CPU-type` option, or the instruction sets supported on the CPU on which GCC was built if the `-mcpu=CPU-type` option was not specified.

`-mno-soft-float`: Specifying this option tells GCC to use hardware floating-point instructions for floating-point operations. This is the default.

`-msmall-data`: Specifying this option causes GCC to store objects that are 8 bytes long or smaller in a small data area (the sdata and sbss sections). These sections are accessed through 16-bit relocations based on the `$gp` register. This limits the size of the small data area to 64K, but allows variables to be directly accessed using a single instruction. This option can only be used when the `-explicit-relocs` option has also been specified.

Note When generating code for shared libraries on an Alpha, specifying the `-fpic` option implies the `-msmall-data` option.

`-msmall-text`: Specifying this option causes GCC to assume that the size of the code produced by the compiler is smaller than 4MB and that therefore any part of the resulting code can be reached by a single branch instruction. This reduces the number of instructions required to make any function call, thereby reducing overall code size.

`-msoft-float`: Specifying this option tells GCC to generate output that contains library calls for floating-point operations. These libraries are not provided as part of GCC, but are normally found on the target system and should be resolved by the C loader on the target machine if they are available. When using this option and cross-compiling, you must provide suitable libraries containing at least function stubs on the host system, or these function calls will be identified as unresolved.

Note Ironically, Alpha implementations without floating-point operations are still required to have floating-point registers.

`-mtls-kernel`: Specifying this option tells GCC that the system for which it is generating code uses the OSF/1 PAL code, and should therefore use the `rduniq` and `wruniq` calls for thread pointer built-ins rather than the `rdval` and `wrval` calls that are normally used.

`-mtls-size=`*`number`*: Specifying this option enables you to tell GCC the bit size of offsets used for thread-local storage, where *`number`* is that value.

`-mtrap-precision=`*`trap-precision`*: Floating-point traps are imprecise in the Alpha architecture; without software assistance, it is impossible to recover from a floating trap, and programs that trap must usually be terminated. Specifying this option causes GCC to generate code that can help operating system trap handlers determine the exact location that caused a floating-point trap. The following levels of precision can be specified as the value of *`trap-precision`*:

- `f`: Function precision. The trap handler can determine the function that caused a floating-point exception.
- `i`: Instruction precision. The trap handler can determine the exact instruction that caused a floating-point exception.
- `p`: Program precision. The trap handler can only identify which program caused a floating-point exception. This is the default.

`-mtune=`*`CPU-type`*: Specifying this option tells GCC to set only the instruction scheduling parameters based on the specified `CPU-type`. The instruction set is not changed. Possible values for *`CPU-type`* are the same as those that can be specified using the `-mcpu=`*`CPU-type`* option.

Alpha/VMS Options

Many workstations using Alpha processors from DEC (and later Compaq Computer or Hewlett-Packard) shipped with a quaint operating system called VMS. For years, VMS was the only operating system for DEC computers for which support was officially available from DEC.

The sole GCC option available when compiling code for Alpha systems running VMS is the following:

`-mvms-return-codes`: Specifying this option causes GCC to return VMS error codes from `main`, rather than the default POSIX-style error codes used by the majority of the known universe.

AMD x86-64 Options

The options in this section can only be used when compiling code targeted for 64-bit AMD processors (the GCC x86-64 build target). For the discussion of options that can only be used on i386 and AMD x86-64 systems, see the section of this appendix titled "i386 and AMD x86-64 Options." For the discussion of options that can only be used on IA-64 (64-bit Intel) systems, see the section of this appendix titled "IA-64 Options."

GCC options available when compiling code for 64-bit AMD systems are the following:

`-m32`: Specifying this option tells GCC to generate code for a 32-bit environment. The 32-bit environment sets `int`, `long`, and `pointer` to 32 bits and generates code that runs on any i386 system.

`-m64`: Specifying this option tells GCC to generate code for a 64-bit environment. The 64-bit environment sets `int` to 32 bits and `long` and `pointer` to 64 bits and generates code for AMD's x86-64 architecture.

`-mcmodel=kernel`: Specifying this option tells GCC to generate code for the kernel code model. The kernel runs in the negative 2GB of the address space. This model must be used for Linux kernel code.

`-mcmodel=large`: Specifying this option tells GCC to generate code for the large code model. This model makes no assumptions about addresses and sizes of sections. This option is reserved for future expansion—GCC does not currently implement this model.

`-mcmodel=medium`: Specifying this option tells GCC to generate code for the medium code model. This means that the program is linked in the lower 2GB of the address space, but symbols can be located anywhere in the address space. Programs can be statically or dynamically linked, but building shared libraries is not supported by this memory model.

`-mcmodel=small`: Specifying this option tells GCC to generate code for the small code model. This means that the program and its symbols must be linked in the lower 2GB of the address space, pointers are 64 bits, and programs can be statically or dynamically linked. This is the default code model.

`-mno-red-zone`: Specifying this option tells GCC not to use a so-called *red zone* for x86-64 code. The red zone is mandated by the x86-64 ABI, and is a 128-byte area beyond the location of the stack pointer that will not be modified by signal or interrupt handlers and can therefore be used for temporary data without adjusting the stack pointer. Using the red zone is enabled by default.

AMD 29K Options

The AMD 29000 processor is a RISC microprocessor descended from the Berkeley RISC design and includes a memory-management unit (MMU) as well as support for the AMD 29027 floating-point unit (FPU). Like conceptually similar processors such as the SPARC, the 29000 has a large register set split into local and global sets and provides sophisticated mechanisms for protecting, manipulating, and managing registers and their contents. The AMD 29000 family of processors includes the 29000, 29005, 29030, 29035, 29040, and 29050 microprocessors.

■ **Note** Support for this processor family was marked as obsolete in GCC 3.1 and was fully purged in GCC version 3.2.2. These options are therefore only of interest if you are using a version of GCC that is earlier than 3.2.1 and that you are certain provides support for this processor family.

GCC options available when compiling code for the AMD AM29000 family of processors are the following:

`-m29000`: Specifying this option causes GCC to generate code that only uses instructions available in the basic AMD 29000 instruction set. This is the default.

`-m29050`: Specifying this option causes GCC to generate code that takes advantage of specific instructions available on the AMD 29050 processor.

`-mbw`: Specifying this option causes GCC to generate code that assumes the system supports byte and half-word write operations. This is the default.

`-mdw`: Specifying this option causes GCC to generate code that assumes the processor's DW bit is set, which indicates that byte and half-word operations are directly supported by the hardware. This is the default.

`-mimpure-text`: Specifying this option in conjunction with the `-shared` option tells GCC not to pass the `-assert` pure-text option to the linker when linking a shared object.

`-mkernel-registers`: Specifying this option causes GCC to generate references to registers `gr64` through `gr95` instead of to registers `gr96` through `gr127` (the latter is the default). This option is often used when compiling kernel code that wants to reserve and use a set of global registers that are disjointed from the set used by user-mode code. Any register names passed as compilation options using GCC's `-f` option must therefore use the standard user-mode register names.

`-mlarge`: Specifying this option causes GCC to always use calli instructions, regardless of the size of the output file. You should use this option if you expect any single file to compile into more than 256K of code.

`-mnbw`: Specifying this option causes GCC to generate code that assumes the processor does not support byte and half-word operations. Specifying this option automatically sets the related `-mndw` option.

`-mndw`: Specifying this option causes GCC to generate code that assumes the processor's DW bit is not set, indicating that byte and half-word operations are not directly supported by the hardware. This option is automatically set if you specify the `-mnbw` option.

`-mno-impure-text`: Specifying this option tells GCC to pass the `-assert` pure-text option to the linker when linking a shared object.

`-mno-multm`: Specifying this option causes GCC not to generate multm or multmu instructions. This option is used when compiling for 29000-based embedded systems that do not have trap handlers for these instructions.

`-mno-reuse-arg-regs`: Specifying this option tells GCC not to reuse incoming argument registers for copying out arguments.

`-mno-soft-float`: Specifying this option tells GCC to use hardware floating-point instructions for floating-point operations. This is the default.

`-mno-stack-check`: Specifying this option causes GCC not to insert a call to `__msp_check` after each stack adjustment.

`-mno-storem-bug`: Specifying this option causes GCC to not generate code that keeps mtsrim, insn, and storem instructions together. This option should be used when compiling for the 29050 processor, which can handle the separation of these instructions.

`-mnormal`: Specifying this option causes GCC to use the normal memory model that generates call instructions only when calling functions in the same file, and generates calli instructions otherwise. This memory model works correctly if each file occupies less than 256K but allows the entire executable to be larger than 256K. This is the default.

`-mreuse-arg-regs`: Specifying this option tells GCC to use only incoming argument registers for copying out arguments. This helps detect functions that are called with fewer arguments than they were declared with.

`-msmall`: Specifying this option causes GCC to use a small memory model that assumes that all function addresses are either within a single 256K segment or at an absolute address of less than 256K. This allows code to use the call instruction instead of a `const`, `consth`, and `calli` sequence.

`-msoft-float`: Specifying this option tells GCC to generate output that contains library calls for floating-point operations. These libraries are not provided as part of GCC but are normally found on the target system and should be resolved by the C loader on the target machine if they are available. When using this option and cross-compiling, you must provide suitable libraries containing at least function stubs on the host system or these function calls will be identified as unresolved.

`-mstack-check`: Specifying this option causes GCC to insert a call to `__msp_check()` after each stack adjustment. This option is often used when compiling kernel or general operating system code.

`-mstorem-bug`: Specifying this option causes GCC to generate code for AMD 29000 processors that cannot handle the separation of mtsrim, insn, and storem instructions. This option should be used on most 29000 processors, with the exception of the 29050.

`-muser-registers`: Specifying this option causes GCC to use the standard set of global registers (`gr96` through `gr127`). This is the default.

ARC Options

The ARC processor is a highly customizable 32-bit RISC processor core that is often used in embedded systems.

GCC options available when compiling code for ARC processors are the following:

`-EB`: Specifying this option causes GCC to generate code that is compiled for big endian mode (where the most significant byte of a word has the lowest address—the word is stored big end first).

`-EL`: Specifying this option causes GCC to generate code that is compiled for little endian mode (where the most significant byte of a word has the highest significance—the word is stored little end first). This is the default.

`-mcpu=`*`CPU`*: Specifying this option causes GCC to generate code compiled for the specific ARC variant named *`CPU`*. The variants supported by GCC depend on the GCC configuration. Possible values are `arc`, `arc5`, `arc6`, `arc7`, `arc8`, and `base`. All ARC variants support `-mcpu=base`, which is the default.

`-mdata=`*`data-section`*: Specifying this option causes GCC to store data in the section whose name is specified as *data-section*. This command-line option can be overridden for specific functions by using the `__attribute__` keyword to set the `section` attribute for those functions, as explained in Appendix A.

`-mmangle-cpu`: Specifying this option causes GCC to add the name of the CPU at the beginning of all public symbol names. Many multiple-processor systems use ARC CPU variants with different instruction and register set characteristics. Using this option prevents code compiled for one CPU from being linked with code compiled for another.

`-mrodata=`*`readonly-data-section`*: Specifying this option causes GCC to store read-only data in the section whose name is specified as *readonly-data-section*. This command-line option can be overridden for specific functions by using the `__attribute__` keyword to set the `section` attribute for those functions, as explained in Appendix A.

`-mtext=`*`text-section`*: Specifying this option causes GCC to put functions in the section whose name is specified as *text-section*, rather than in the default text section. This command-line option can be overridden for specific functions by using the `__attribute__` keyword to set the `section` attribute for those functions, as explained in Appendix A.

ARM Options

The Advanced RISC Machines (ARM) processor is a 32-bit processor family that is extremely popular in embedded, low-power systems. The ARM acronym originally stood for Acorn RISC Machine because the processor was originally designed by Acorn Computer Systems. Advanced RISC Machines Ltd. was formed to market and develop the ARM processor family, related chips, and associated software, at which point the more modern acronym expansion was adopted.

■ **Note** The ARM instruction set is a complete set of 32-bit instructions for the ARM architecture. The Thumb instruction set is an extension to the 32-bit ARM architecture that provides very high code density through a subset of the most commonly used 32-bit ARM instructions that have been compressed into 16-bit-wide operation codes. On execution, these 16-bit instructions are decoded to enable the same functions as their full 32-bit ARM instruction equivalents.

The ARM Procedure Call Standard (APCS) is frequently referenced in ARM-oriented command-line options for GCC. APCS is a set of standards that defines the use of registers on ARM processors, conventions for using the stack, mechanisms for passing and returning arguments across function calls, and the format and use of the stack.

GCC options available when compiling code for ARM processors are the following:

`-mabi=`*name*: Specifying this option tells GCC to generate code for the specified ARM Application Binary Interface (ABI). Possible values for *name* are `apcs-gnu` (the 26- and 32-bit versions of the old APCS), `atpcs` (ARM/Thumb Procedure Call Standard), `aapcs` (ARM Architecture Procedure Call Standard), `aapcs-linux` (ARM Architecture Procedure Call Standard for Linux), and `iwmmxt` (the ABI used on ARM processors that support Intel's wireless MMX technology).

`-mabort-on-noreturn`: Specifying this option causes GCC to generate a call to the `abort` function at the end of each `noreturn` function. The call to `abort` is executed if the function tries to return.

`-malignment-traps`: Specifying this option causes GCC to generate code that will not trap if the MMU has alignment traps enabled by replacing unaligned accesses with a sequence of byte accesses. This option is only relevant for ARM processors prior to the ARM 4 and is ignored on later processors because these have instructions to directly access half-word objects in memory. This option is not supported in any GCC 4.x compiler.

ARM architectures prior to ARM 4 had no instructions to access half-word objects stored in memory. However, a feature of the ARM architecture allows a word load to be used when reading from memory even if the address is unaligned, because the processor core rotates the data as it is being loaded. Specifying this option tells GCC that such misaligned accesses will cause an MMU trap and that it should replace the misaligned access with a series of byte accesses. The compiler can still use word accesses to load half-word data if it knows that the address is aligned to a word boundary.

`-mapcs`: Specifying this option is the same as specifying the `-mapcs-frame` option.

`-mapcs-26`: Specifying this option causes GCC to generate code for an ARM processor running with a 26-bit program counter. The generated code conforms to calling standards for the APCS 26-bit option. The `-mapcd-26` option replaces the `-m2` and `-m3` options provided in older releases of GCC. The `–mapcs-26` option itself is obsolete in the GCC 4.x family of compilers, having largely been replaced by the `-mabi=apcs-gnu` option.

`-mapcs-32`: Specifying this option causes GCC to generate code for an ARM processor running with a 32-bit program counter. The generated code conforms to calling standards for the APCS 32-bit option. The `-mapcd-26` option replaces the `-m6` option provided in older releases of GCC. The `-mapcs-32` option itself is obsolete in the GCC 4.x family of compilers, having largely been replaced by the `-mabi=apcs-gnu` option.

`-mapcs-float`: Specifying this option tells GCC to pass floating-point arguments in floating-point registers.

`-mapcs-frame`: Specifying this option causes GCC to generate a stack frame that is compliant with the APCS for all functions, even if this is not strictly necessary for correct execution of the code. Using this option in conjunction with the `-fomit-frame-pointer` option causes GCC not to generate stack frames for leaf functions (functions that do not call other functions). The default is to not generate stack frames.

`-mapcs-reentrant`: Specifying this option tells GCC to generate re-entrant, position-independent code.

`-mapcs-stack-check`: Specifying this option causes a stack limit of 64K to be set in the code produced by GCC, with tests imposed to ensure that this stack limit is not exceeded. Used primarily in bare metal ARM code—that is, code that is designed to run on ARMs without an operating system.

`-march=`*`name`*: Using this option enables you to identify the ARM architecture used on the system for which you are compiling. Like the `-mcpu` option, GCC uses this name to determine the instructions that it can use when generating assembly code. This option can be used in conjunction with or instead of the `-mcpu=` option. Possible values for *name* are `armv2`, `armv2a`, `armv3`, `armv3m`, `armv4`, `armv4t`, `armv5`, `armv5t`, `armv5te`, `armv6`, `armv6j`, `iwmmxt`, and `ep9312`. See the `-mcpu=`*`name`* and `-mtune=`*`name`* options for related information.

`-marm`: Specifying this option identifies the system as an ARM system without Thumb support.

`-mbig-endian`: Specifying this option causes GCC to generate code for an ARM processor running in big endian mode.

`-mbsd`: Specifying this option causes GCC to emulate the native BSD-mode compiler. This is the default if `-ansi` is not specified. This option is only relevant when compiling code for RISC iX, which is Acorn's version of Unix that was supplied with some ARM-based systems such as the Acorn Archimedes R260. This option is no longer supported in the GCC 4.x compilers.

`-mcallee-super-interworking`: Specifying this option causes GCC to insert an ARM instruction set header before executing any externally visible function. *Interworking mode* is a mode of operation in which ARM and Thumb instructions can interact. This header causes the processor to switch to Thumb mode before executing the rest of the function. This allows these functions to be called from noninterworking code.

`-mcaller-super-interworking`: Specifying this option enables calls via function pointers (including virtual functions) to execute correctly regardless of whether the target code has been compiled for interworking or not. This option causes a slight increase in the cost of executing a function pointer but facilitates interworking in all circumstances.

`-mcirrus-fix-invalid-isns`: Specifying this option (with one of the best names I've ever seen) causes GCC to insert NOOPs in the code that it generates to avoid invalid combinations and sequences of operations in some ARM-based products from Cirrus Logic.

`-mcpu=`*name*: Using this option enables you to specify the particular type of ARM processor used on the system for which you are compiling. GCC uses this name to determine the instructions that it can use when generating assembly code. Possible values for *name* are `arm2`, `arm250`, `arm3`, `arm6`, `arm60`, `arm600`, `arm610`, `arm620`, `arm7`, `arm7m`, `arm7d`, `arm7dm`, `arm7di`, `arm7dmi`, `arm70`, `arm700`, `arm700i`, `arm710`, `arm710c`, `arm7100`, `arm7500`, `arm7500fe`, `arm7tdmi`, `arm8`, `strongarm`, `strongarm110`, `strongarm1100`, `arm8`, `arm810`, `arm9`, `arm9e`, `arm920`, `arm920t`, `arm940t`, `arm9tdmi`, `arm10tdmi`, `arm1020t`, `arm1136j-s`, `arm1136jf-s`, `arm1176jz-s`, `arm1176jzf-s`, `ep9312`, `iwmmxt`, `mpcore`, `mpcorenovfp`, and `xscale`. See the `-march=`*name* and `-mtune=`*name* options for related information.

■ **Note** Because ARM licenses its cores to different manufacturers and is constantly developing new processors, the previous list is likely to change by the time that you read this.

`-mfloat-abi=`*name*: Using this option provides a single centralized mechanism for identifying how GCC should generate floating-point instructions. Valid values for *name* are `soft` (generates floating-point instructions as library calls, equivalent to `-msoft-float`), `hard` (generates actual floating-point instructions, equivalent to `-mhard-float`), and `softfp` (generates actual floating point instructions but uses the `-msoft-float` calling conventions). The older options are still supported.

`-mfp=`*number* | `-mfpe=`*number* | `-mfpu=`*name*: Using these options enables you to specify the version of floating-point emulation that is available on the system for which you are compiling. Possible values for *number* are `2` and `3`, which identify different internal implementations. Possible values for *name* are `fpa`, `fpe2`, `fpe3`, `maverick`, and `vfp`. You should use `-mpfu=`*name* (the most modern of these options) in any new scripts or Makefiles, as the other options are technically obsolete; `-mfp=2` and `-mfpe=2` are now internal aliases for `-mfpu=fpe2`, while `-mfp=3` and `-mfpe=3` are now aliases for `-mfpu=fpe3`.

`-mhard-float`: Specifying this option causes GCC to generate output containing floating-point instructions, which is the default. This option still works, but has been conceptually replaced by `-mfloat-abi=hard`.

`-mlittle-endian`: Specifying this option causes GCC to generate code for an ARM processor running in little endian mode. This is the default.

`-mlong-calls`: Specifying this option causes GCC to perform function calls by first loading the address of the function into a register and then performing a subroutine call on this register. This option is necessary if the target function will lie outside of the 64MB addressing range of the offset-based version of the subroutine call instruction. Specifying this option does not affect calls to static functions, functions that have the `short-call` attribute, functions that are inside the scope of a `#pragma no_long_calls` directive, and functions whose definitions have already been compiled within the current compilation unit. Specifying this option does affect weak function definitions, functions with the `long-call` attribute or the `section` attribute, and functions that are within the scope of a `#pragma long_calls` directive.

`-mno-alignment-traps`: Specifying this option causes GCC to generate code that assumes that the MMU will not trap unaligned accesses. This produces better code for ARM processors prior to the ARM 4, where the target instruction set does not have half-word memory access operations. Unfortunately, you cannot use this option to access unaligned word objects, since the processor will only fetch one 32-bit aligned object from memory. This option is not supported in the GCC 4.x family of compilers.

`-mno-apcs-frame`: Specifying this option causes GCC not to generate stack frames that are compliant with the ARM Procedure Call Standard for all functions unless they are necessary for correct execution of the code. This option is the default.

`-mno-long-calls`: Specifying this option causes GCC to perform function calls in the standard fashion, rather than by the mechanism described in the `-mlong-calls` option. This is the default.

`-mno-sched-prolog`: Specifying this option causes GCC to suppress possible optimizations, preventing reordering instructions in the function prologue and preventing merging those instructions with the instructions in the function's body. Specifying this option means that all functions will start with a recognizable set of instructions from one of the sets of default function prologues, which can then be used to identify the beginning of functions within an executable piece of code. Using this option typically results in larger executables than the default option, `-msched-prolog`.

`-mno-short-load-bytes`: This is a deprecated alias for the `-mno-alignment-traps` option.

`-mno-short-load-words`: This is a deprecated alias for the `-malignment-traps` option.

`-mno-soft-float`: Specifying this option tells GCC to use hardware floating-point instructions for floating-point operations. This option implies `-mhard-float`, but should not be used in new code for logical reasons, because the new `-mfloat-abi=`*name* option provides three options, and therefore *not soft-float* can mean one of two options to anyone who inherits your code.

`-mno-symrename`: Specifying this option causes GCC not to run the assembler post-processor, symrename, after assembling code on a RISC iX system. This post-processor is normally run in order to modify standard symbols in assembly output so that resulting binaries can be successfully linked with the RISC iX C library. This option is only relevant for versions of GCC that are running directly on a RISC iX system: no such post-processor is provided by GCC when GCC is used as a cross-compiler. This option is not supported by the GCC 4.x family of compilers, because RISC iX systems are not supported by GCC 4.x.

`-mno-thumb-interwork`: Specifying this option causes GCC to generate code that does not support calls between the ARM and Thumb instruction sets. This option is the default. See the `-mthumb-interwork` option for related information.

`-mno-tpcs-frame`: Specifying this option causes GCC not to generate stack frames that are compliant with the Thumb Procedure Call Standard. This option is the default. See the `-mtpcs-frame` option for related information.

`-mno-tpcs-leaf-frame`: Specifying this option causes GCC to not generate stack frames that are compliant with the Thumb Procedure Call Standard. This option is the default. See the `-mtpcs-leaf-frame` option for related information.

`-mnop-fun-dllimport`: Specifying this option causes GCC to disable support for the `dllimport` attribute.

`-mpic-register=`*name*: Using this option enables you to specify the register (`name`) to be used for PIC addressing. When this option is not supplied, the default register used for PIC addressing is register `R10` unless stack checking is enabled, in which case register `R9` is used.

`-mpoke-function-name`: Specifying this option causes GCC to write the name of each function into the text section immediately preceding each function prologue. Specifying this option means that all functions will be preceded by a recognizable label that can then be used to identify the beginning of functions within an executable piece of code.

`-msched-prolog`: Specifying this option causes GCC to optimize function prologue and body code sequences, merging operations whenever possible. This option is the default.

`-mshort-load-bytes`: This is a deprecated alias for the `-malignment-traps` option, which itself is not supported in the GCC 4.x family of compilers.

`-mshort-load-words`: This is a deprecated alias for the `-mno-alignment-traps` option, which itself is not supported in the GCC 4.x family of compilers.

`-msingle-pic-base`: Specifying this option causes GCC to treat the register used for PIC addressing as read-only, rather than loading it in the prologue for each function. The runtime system is responsible for initializing this register with an appropriate value before execution begins.

`-msoft-float`: Specifying this option causes GCC to generate output containing library calls for floating point. These libraries are not provided as part of GCC but are normally found on the target system and should be resolved by the C loader on the target machine if they are available. When using this option and cross-compiling, you must provide suitable libraries containing at least function stubs on the host system, or these function calls will be identified as unresolved. This option still works but has been conceptually replaced by `-mfloat-abi=soft`.

■ **Tip** Specifying this option also changes the calling convention used in the output file. You must therefore compile all of the modules of your program with this option, including any libraries that you reference. You must also compile libgcc.a, the library that comes with GCC, with this option in order to be able to use it in your applications.

`-mstructure-size-boundary=`*n*: Using this option causes GCC to round the size of all structures and unions up to a multiple of the number of bits (*n*) specified with this option. Valid values are `8` and `32`, and vary for different output formats; the default value is 8 for COFF output toolchains. Code compiled with one value will not necessarily work with code or libraries compiled with the other value. Specifying the larger number might produce faster, more efficient code, but may also increase the size of the program.

`-mthumb`: Specifying this option causes GCC to generate code for the 16-bit Thumb instruction set. The default is to use the 32-bit ARM instruction set.

`-mthumb-interwork`: Specifying this option causes GCC to generate code that supports calls between the ARM and Thumb instruction sets. If this option is not specified, the two instruction sets cannot be reliably used inside one program. Specifying this option causes GCC to generate slightly larger executables than the ARM-specific default option, `-mno-thumb-interwork`.

`-mtpcs-frame`: Specifying this option causes GCC to generate stack frames that are compliant with the Thumb Procedure Call Standard. Specifying this option only affects nonleaf functions (i.e., functions that call other functions). The default is `-mno-tpcs-frame`—in other words, not to generate compliant stack frames.

`-mtpcs-leaf-frame`: Specifying this option causes GCC to generate stack frames that are compliant with the Thumb Procedure Call Standard. Specifying this option only affects leaf functions (i.e., functions that do not call other functions). The default is `-mno-tpcs-leaf-frame`—in other words, not to generate compliant stack frames.

`-mtune=`*`name`*: Specifying this option causes GCC to tune the generated code as though the target ARM processor were of type *name*, but to still generate code that conforms to the instructions available for an ARM processor specified using the `-mcpu` option. Using these two options together can provide better performance on some ARM-based systems. See the `-march=`*`name`* and `-mcpu=`*`name`* options for related information.

`-mwords-little-endian`: Specifying this option causes GCC to generate code for a little endian word order but a big endian byte order (i.e., of the form 32107654). This option only applies when generating code for big endian processors and should only be used for compatibility with code for big endian ARM processors that was generated by versions of GCC prior to 2.8.

`-mxopen`: Specifying this option causes GCC to emulate the native X/Open-mode compiler. This is the default if `-ansi` is not specified. This option is only relevant when compiling code for RISC iX, which is Acorn's version of Unix that was supplied with some ARM-based systems such as the Acorn Archimedes R260. This option is not supported in the GCC 4.x compiler family, because RISC iX systems are no longer supported.

AVR Options

Atmel's AVR processors are microcontrollers with a RISC core running single-cycle instructions. They provide a well-defined I/O structure that limits the need for external components, and are therefore frequently used in embedded systems.

GCC options available when compiling code for AVR processors are the following:

`-mcall-prologues`: Specifying this option causes GCC to expand function prologues and epilogues as calls to the appropriate subroutines, reducing code size.

`-mdeb`: Specifying this option generates extensive debugging information.

`-minit-stack=`*`n`*: Specifying this option enables you to define the initial stack address, which may be a symbol or a numeric value. The value `__stack` is the default.

`-mint8`: Specifying this option tells GCC to use an 8-bit `int` datatype. When this option is used, a `char` will also be 8 bits, a `long` will be 16 bits, and a `long long` will be 32 bits. Though this option does not conform to C standards, using it will almost always produce smaller binaries.

`-mmcu=`*`MCU`*: Specifying this option enables you to specify the AVR instruction set or `MCU` type for which code should be generated. Possible values for the instruction set are the following:

- `avr1`: The minimal AVR core. This value is not supported by the GCC C compiler but only by the GNU assembler. Associated `MCU` types are `at90s1200`, `attiny10`, `attiny11`, `attiny12`, `attiny15`, and `attiny28`.
- `avr2`: The classic AVR core with up to 8K of program memory space. This is the default. Associated `MCU` types are `at90s2313`, `at90s2323`, `attiny22`, `at90s2333`, `at90s2343`, `at90s4414`, `at90s4433`, `at90s4434`, `at90s8515`, `at90c8534`, and `at90s8535`.
- `avr3`: The classic AVR core with up to 128K of program memory space. Associated `MCU` types are `atmega103`, `atmega603`, `at43usb320`, and `at76c711`.
- `avr4`: The enhanced AVR core with up to 8K of program memory space. Associated `MCU` types are `atmega8`, `atmega83`, and `atmega85`.
- `avr5`: The enhanced AVR core with up to 128K of program memory space. Associated `MCU` types are `atmega16`, `atmega161`, `atmega163`, `atmega32`, `atmega323`, `atmega64`, `atmega128`, `at43usb355`, and `at94k`.

`-mno-interrupts`: Specifying this option causes GCC to generate code that is not compatible with hardware interrupts, reducing code size.

`-mno-tablejump`: Specifying this option causes GCC not to generate tablejump instruction, which may increase code size.

`-mshort-calls`: Specifying this option causes GCC to use the rjmp/rcall instructions (which have limited range) on devices with greater than 8K of memory.

`-msize`: Specifying this option causes GCC to output instruction sizes to the assembler output file.

`-mtiny-stack`: Specifying this option tells GCC to generate code that only uses the low 8 bits of the stack pointer.

Blackfin Options

Analog Devices' Blackfin processors are very interesting 32-bit RISC processors that are primarily designed for audio, video, communications, and related embedded projects, offering a complete 32-bit RISC programming model on an SIMD (single-instruction, multiple-data stream) architecture.

GCC options available when compiling code for Blackfin processors are the following:

`-mcsync-anomaly`: Specifying this option tells GCC to generate extra code that guarantees that CSYNC or SSYNC instructions do not occur too soon after any conditional branch. This option works around a hardware anomaly and is active by default.

`-mid-shared-library`: Specifying this option tells GCC to generate code that supports shared library using library IDs, and implies the generic GCC `-fPIC` option. This option therefore supports things such as execute in place and shared libraries without requiring virtual memory management.

`-mlong-calls`: Specifying this option tells GCC to perform function calls by first loading the address of the function into a register and then performing a subroutine call on this register. This option is necessary if the target function will lie outside of the 24-bit addressing range of the offset-based version of the subroutine call instruction.

`-mlow64k`: Specifying this option tells GCC that the entire program will fit into the low 64K of memory and to perform any possible optimizations related to that fact.

`-mno-csync-anomaly`: Specifying this option tells GCC not to generate any extra NOOPs necessary to prevent CSYNC or SSYNC instructions from occurring too quickly after conditional branches.

`-mno-id-shared-library`: Specifying this option tells GCC not to generate code that supports shared libraries using library IDs. This option is active by default.

`-mno-long-calls`: Specifying this option causes GCC to perform function calls in the standard fashion, rather than by the mechanism described in the `-mlong-calls` option. This is the default.

`-mno-low64k`: Specifying this option tells GCC that the program size is unknown and not to attempt to perform optimizations based on assuming that the program is less than 64K. This option is active by default.

`-mno-omit-leaf-frame-pointer`: Specifying this option tells GCC to retain the frame pointer in a register for leaf functions, simplifying debugging but generating larger binaries due to the setup, save, and restore operations.

`-mno-specld-anomaly`: Specifying this option tells GCC not to generate the extra code used to prevent speculative loads after jumps.

`-momit-leaf-frame-pointer`: Specifying this option tells GCC not to keep the frame pointer in a register for leaf functions. This reduces the number of instructions required to set up, save, and restore frame pointers but makes debugging more complex.

`-mshared-library-id=`*n*: Specifying this option enables you to specify the ID number (*n*) of an ID-based shared library that is being compiled.

`-mspecld-anomaly`: Specifying this option tells GCC to generate extra code to prevent speculative loads after jump instructions. This option works around a hardware anomaly, and is active by default.

Clipper Options

Clipper is a family of RISC processors that were primarily used in older Intergraph Unix workstations.

■ **Note** Support for this processor family was marked as obsolete in GCC 3.1 and was fully purged in GCC version 3.2.2. These options are therefore only of interest if you are using a version of GCC that is earlier than 3.2.1 and that you are certain provides support for this processor family.

GCC options available when compiling code for Clipper processors are the following:

`-mc300`: Specifying this option causes GCC to generate code for a C300 Clipper processor. This is the default.

`-mc400`: Specifying this option causes GCC to generate code for a C400 Clipper processor. The generated code uses floating-point registers `f8` through `f15`.

Convex Options

Convex Computer systems were minisupercomputers that were targeted for use by small to medium-sized businesses. Systems such as the C1, C2, and C3 were high-performance vector-processing systems that were substantially less expensive than the competing systems from Cray Research. The later Exemplar systems were based on the Hewlett-Packard PA-RISC CPU series. Convex was acquired by HP in 1995.

■ **Note** Support for this processor family was marked as obsolete in GCC 3.1 and was fully purged in GCC version 3.2.2. These options are therefore only of interest if you are using a version of GCC that is earlier than 3.2.1 and that you are certain provides support for this processor family.

GCC options available when compiling code for Convex systems are the following:

`-margcount`: Specifying this option causes GCC to generate code that puts an argument count in the word preceding each argument list. This argument-count word is compatible with the regular C compiler provided by ConvexOS, and may be needed by some programs.

`-mc1`: Specifying this option causes GCC to generate code targeted for Convex C1 systems, but which will run on any Convex machine. This option defines the preprocessor symbol `__convex_c1__`.

`-mc2`: Specifying this option causes GCC to generate code targeted for Convex C2 systems, but which should also run on Convex C3 machines, though scheduling and other optimizations are chosen for maximum performance on C2 systems. This option defines the preprocessor symbol `__convex_c2__`.

`-mc32`: Specifying this option causes GCC to generate code targeted for Convex C32xx systems, using scheduling and other optimizations chosen for maximum performance on C32xx systems. This option defines the preprocessor symbol `__convex_c32__`.

`-mc34`: Specifying this option causes GCC to generate code targeted for Convex C34xx systems, using scheduling and other optimizations chosen for maximum performance on C34xx systems. This option defines the preprocessor symbol `__convex_c34__`.

`-mc38`: Specifying this option causes GCC to generate code targeted for Convex C38xx systems, using scheduling and other optimizations that are chosen for maximum performance on C38xx systems. This option defines the preprocessor symbol `__convex_c38__`.

`-mlong32`: Specifying this option causes GCC to define type `long` as 32 bits, the same as type `int`. This is the default.

`-mlong64`: Specifying this option causes GCC to define type `long` as 64 bits, the same as type `long long`. This option is essentially useless because there is no support for this convention in the GCC libraries.

`-mnoargcount`: Specifying this option causes GCC to omit the argument-count word. This is the default. See the `-margcount` option for related information.

`-mvolatile-cache`: Specifying this option causes GCC to generate code in which volatile references are cached. This is the default.

`-mvolatile-nocache`: Specifying this option causes GCC to generate code in which volatile references bypass the data cache, going directly to memory. This option is only needed for multiprocessor code that does not use standard synchronization instructions. Making nonvolatile references to volatile locations will not necessarily work.

CRIS Options

Axis Solutions' Code Reduced Instruction Set (CRIS) processors are frequently used in network-oriented embedded applications.

GCC options available when compiling code for CRIS systems are the following:

`-m16-bit`: Specifying this option tells GCC to align the stack frame, writable data, and constants to all be 16-bit aligned. The default is 32-bit alignment.

`-m32-bit`: Specifying this option tells GCC to align the stack frame, writable data, and constants to all be 32-bit aligned. This is the default.

`-m8-bit`: Specifying this option tells GCC to align the stack frame, writable data, and constants to all be 8-bit aligned. The default is 32-bit alignment.

`-maout`: This deprecated option is a NOOP that is only recognized with the `cris-axis-aout` GCC build target.

`-march=`*`architecture-type`* | `-mcpu=`*`architecture-type`*: Specifying either of these options causes GCC to generate code for the specified architecture. Possible values for *`architecture-type`* are `v3` (for ETRAX 4), `v8` (for ETRAX 100), and `v10` (for ETRAX 100 LX). The default is `v0` except for the `cris-axis-linux-gnu` GCC build target, where the default is `v10`.

`-mbest-lib-options`: Specifying this option tells GCC to select the most feature-rich set of options possible based on all other options that have been specified.

`-mcc-init`: Specifying this option tells GCC not to use condition-code results from previous instructions, but to always generate compare and test instructions before using condition codes.

`-mconst-align`: Specifying this option tells GCC to align constants for the maximum single data access size for the chosen CPU model. The default is to arrange for 32-bit alignment. ABI details such as structure layout are not affected by this option.

`-mdata-align`: Specifying this option tells GCC to align individual data for the maximum single data access size for the chosen CPU model. The default is to arrange for 32-bit alignment. ABI details such as structure layout are not affected by this option.

`-melf`: This deprecated option is a NOOP that is only recognized with the `cris-axis-elf` and `cris-axis-linux-gnu` GCC build targets.

`-melinux`: Specifying this option tells GCC to select a GNU/Linux-like multilib, using include files and the instruction set for `-march=v8`. This option is only recognized with the `cris-axis-aout` GCC build target.

`-melinux-stacksize=`*`n`*: Specifying this option tells GCC to include instructions in the program so that the kernel loader sets the stack size of the program to *n* bytes. This option is only available on the `cris-axis-aout` GCC build target.

`-metrax100`: Specifying this option is the same as the `-march=v8` option.

`-metrax4`: Specifying this option is the same as the `-march=v3` option.

`-mgotplt`: Specifying this option, when used with the `-fpic` or `-fPIC` options, tells GCC to generate instruction sequences that load addresses for functions from the PLT (procedure linkage table) part of the GOT (global offset table) rather than by making calls to the PLT. The GOT holds the resolved virtual addresses of symbols. The PLT provides the glue between a function call and the dynamically linked target of that call. This option is active by default.

`-mlinux`: This deprecated option is a NOOP that is only recognized with the `cris-axis-linux-gnu` GCC build target.

`-mmax-stack-frame=`*`n`*: Specifying this option causes GCC to display a warning when the stack frame of a function exceeds *n* bytes.

`-mmul-bug-workaround`: Specifying this option tells GCC to generate additional code to work around a hardware bug in multiplication instructions on CRIS processors, such as the ETRAX 100, where the problem may be present.

`-mno-const-align`: Specifying this option tells GCC not to align the constants for the maximum single data access size for the chosen CPU model. The default is to arrange for 32-bit alignment. ABI details such as structure layout are not affected by this option.

`-mno-data-align`: Specifying this option tells GCC not to align individual data for the maximum single data access size for the chosen CPU model. The default is to arrange for 32-bit alignment. ABI details such as structure layout are not affected by this option.

`-mno-gotplt`: When used with the `-fpic` or `-fPIC` options, specifying this option tells GCC to make calls to the PLT rather than load addresses for functions from the PLT part of the GOT.

`-mno-mul-bug-workaround`: Specifying this option tells GCC not to generate the additional code required to work around a hardware bug in multiplication instructions on CRIS processors, such as the ETRAX 100, where the problem may be present.

`-mno-prologue-epilogue`: Specifying this option tells GCC to omit the normal function prologue and epilogue that sets up the stack frame, and not to generate return instructions or return sequences in the code. This option is designed to simplify visual inspection of compiled code, as it may generate code that stops on certain registers. No warnings or errors are generated when call-saved registers must be saved, or when storage for local variables needs to be allocated.

`-mno-side-effects`: Specifying this option tells GCC not to emit instructions with side effects when using addressing modes other than post-increment.

`-mno-stack-align`: Specifying this option tells GCC not to align the stack frame for the maximum single data access size for the chosen CPU model. The default is to arrange for 32-bit alignment. ABI details such as structure layout are not affected by this option.

`-moverride-best-lib-options`: Specifying this option tells GCC not to automatically select the most feature-rich set of options possible based on all other options that have been specified. This option is required for some files compiled with multilib support. (This option is not available in the GCC 4.x compilers.)

`-mpdebug`: Specifying this option tells GCC to enable verbose CRIS-specific debugging information in the assembly code. This option also turns off the `#NO_APP` formatted-code indicator at the beginning of the assembly file.

`-mprologue-epilogue`: Specifying this option tells GCC to include the normal function prologue and epilogue that set up the stack frame. This is the default.

`-mstack-align`: Specifying this option tells GCC to align the stack frame for the maximum single data access size for the specified CPU model. The default is to arrange for 32-bit alignment. ABI details such as structure layout are not affected by this option.

`-mtune=`*`architecture-type`*: Specifying this option causes GCC to tune the generated code for the specified `architecture-type`, except for the ABI and the set of available instructions. The choices for *`architecture-type`* are the same as those for the `-march=`*`architecture-type`* option.

`-sim`: Specifying this option tells GCC to link with input-output functions from a simulator library. Code, initialized data, and zero-initialized data are allocated consecutively. This option is only recognized for the `cris-axis-aout` and `cris-axis-elf` GCC build targets.

`-sim2`: Specifying this option tells GCC to link with input-output functions from a simulator library and to pass linker options to locate initialized data at the memory address 0x40000000 and zero-initialized data at 0x80000000. Code, initialized data, and zero-initialized data are allocated consecutively. This option is only recognized for the `cris-axis-aout` and `cris-axis-elf` GCC build targets.

CRX Options

National Semiconductor's CRX microcontrollers are 32-bit RISC processors that are a follow-up to its 16-bit CompactRISC CR16 processors.

GCC options available when compiling code for CRX-based systems are the following:

`-mloopnesting=n`: Specifying this option tells GCC to restrict nested `do` loops to the nesting level specified by *n*. (This option is not available in the GCC 4.x compilers.)

`-mmac`: Specifying this option tells GCC to support the use of the multiply-accumulate instructions. This option is disabled by default.

`-mno-push-args`: Specifying this option tells GCC not to use push instructions to pass outgoing arguments when functions are called.

`-mpush-args`: Specifying this option tells GCC to use push instructions to pass outgoing arguments when functions are called. This option is enabled by default.

D30V Options

Mitsubishi's D30V processor is a RISC processor with integrated hardware support for a real-time MPEG-2 decoder, and therefore targets embedded multimedia applications.

■ **Note** I've never actually seen one of these processors and I believe that GCC was just beginning to be used when Mitsubishi was developing its hardware prototypes. These processors are not supported in the GCC 4.x family of compilers. I would not try to use them with anything later than GCC 3.2. Support for them was dropped long ago in related tools such as GDB 5.2.

GCC options available when compiling code for D30V-based systems are the following:

`-masm-optimize`: Specifying this option tells GCC to pass the `-O` option to the assembler during optimization. The assembler uses the `-O` option to automatically parallelize adjacent short instructions whenever possible.

`-mbranch-cost=n`: Specifying this option tells GCC to increase the internal cost of a branch to the value specified as *n*. Higher costs mean that the compiler will generate more instructions in order to avoid doing a branch. Avoiding branches is a common performance optimization for RISC processors. The default is 2.

`-mcond-exec=n`: Specifying this option enables you to identify the maximum number of conditionally executed instructions that replace a branch as the value *n*. The default is 4.

`-mextmem | -mextmemory`: Specifying these options tells GCC to link the text, data, bss, strings, rodata, rodata1, and data1 sections into external memory, which starts at location 0x80000000.

`-monchip`: Specifying this option tells GCC to link the text section into on-chip text memory, which starts at location 0x0. This option also tells GCC to link data, bss, strings, rodata, rodata1, and data1 sections into on-chip data memory, which starts at location 0x20000000.

`-mno-asm-optimize`: Specifying this option tells GCC to not pass the `-O` option to the assembler, thereby suppressing optimization.

Darwin Options

Darwin is the open source operating system environment that underlies Apple's Mac OS X operating system. Based on a combination of the Mach operating system kernel and technologies from FreeBSD, it is a freely available operating system that has its roots in the Berkeley Standard Distribution of

Unix. As you'll see in this section, Apple has heavily leveraged and customized the GCC tools and related open source technologies such as binutils for use as compilers, linkers, and so on, to compile Darwin, Darwin system utilities that run from the Mac OS X command line, and the graphical components of Mac OS X, known as Aqua. Because GCC, binutils, and so on, are all open source software, we all benefit from Apple's adoption of GCC.

Of course, as this appendix shows, adopting GCC to a new operating system results in many new options for GCC (and especially the Darwin version of ld). The GCC-specific options for Darwin are the following:

`-all_load`: Specifying this option causes GCC to load all members of static archive libraries.

`-arch_errors_fatal`: Specifying this option causes GCC compilers to treat any errors associated with files that were compiled for the wrong architecture as fatal. This option is especially important when producing the new generation of Mac OS X fat (multiarchitecture) binaries that contain both PowerPC and x86 executables.

`-bind_at_load`: Specifying this option causes GCC compilers to mark an object file or executable such that the Mac OS X dynamic linker will bind all undefined references when the file is loaded or executed.

`-bundle`: Specifying this option causes GCC compilers to produce a Mach-O bundle format file. Headers are located in the first segment; all sections are placed in the proper segments and are padded to ensure proper segment alignment. The file type for the resulting bundle is `MH_BUNDLE`.

`-bundle_loader` *`executable`*: Specifying this option identifies the executable that will be loading the object file(s) that are being linked. Undefined symbols in the bundle are checked against the specified executable just as if it were a dynamic library that the bundle was linked with. If the `-twolevel_namespace` option is used to segment the symbol namespace, symbol searches are based on the placement of the `-bundle_loader` option. If the `-flat_namespace` option is specified, the executable is searched before all dynamic libraries. This option is only valid for 32-bit executables.

`-dynamiclib`: Specifying this option causes GCC compilers to produce a dynamic library using the Darwin libtool command rather than an executable when linking.

`-F`*`dir`*: Specifying this option causes GCC compilers to add the frameworks directory *`dir`* to the beginning of the list of directories to be searched for header files. Frameworks is the term for the directory structure and associated set of naming conventions used with Apple's Xcode development environment and tools.

`-force_cpusubtype_ALL`: Specifying this option causes GCC compilers to produce output files that have the `ALL` subtype, a set of instructions common to all processors, rather than the one controlled by the `-mcpu` or `-march` options. This option is especially important when producing the new generation of Mac OS X fat (multiarchitecture) binaries that contain both PowerPC and x86 executables.

`-gused`: Specifying this option causes GCC compilers to generate debugging information only for symbols that are used. This option is on by default. For projects that use the STABS debugging format, specifying this option also enables the `-feliminate-unused-debug-symbols` option.

`-gfull`: Specifying this option causes GCC compilers to generate debugging information for all symbols and types.

`-mfix-and-continue` | `-ffix-and-continue` | `-findirect-data`: Specifying any of these options causes GCC to generate code that is suitable for fast turnaround development, enabling GDB to dynamically load new object files into programs that are already running. The `-findirect-data` and `-ffix-and-continue` synonyms for `-mfix-and-continue` are provided for backward compatibility.

`-mmacosx-version-min=`*`version`*: Specifying this option identifies Mac OS X version *version* as the earliest version of Mac OS X that an executable will run on. The value for *version* can be a two- or three-digit dot-separated version identifier. Common values for *version* are `10.1`, `10.2`, and `10.3.9`.

`-mone-byte-bool`: Specifying this option tells GCC compilers to use a single byte to store Boolean values, so that `sizeof(bool)` is 1. This option is only meaningful on Darwin/PowerPC platforms where `sizeof(bool)` is normally 4. Using this option generates object code that is incompatible with object code generated without specifying this option. You will therefore need to use this option with none or all of the object code involved in an executable, including system libraries.

■ **Note** GCC for the Darwin platform also supports a number of options that it simply passes through to the Darwin linker in order to invoke specific linker options. Since these are actually options for the Darwin linker rather than the vanilla GNU linker, they are not discussed here—for details, see the man page for ld, the GNU linker/loader, on a Darwin/Mac OS X system. These options are the following: `-allowable_client` *`CLIENT-NAME`*, `-client_name` *`CLIENT-NAME`*, `-compatibility_version` *`NUMBER`*, `-current_version` *`NUMBER`*, `-dead_strip`, `-dependency-file`, `-dylib_file`, `-dylinker_install_name` *`NAME`*, `-dynamic`, `-exported_symbols_list`, `-filelist`, `-flat_namespace`, `-force_flat_namespace`, `-headerpad_max_install_names`, `-image_base`, `-init`, `-install_name` *`NAME`*, `-keep_private_externs`, `-multi_module`, `-multiply_defined`, `-multiply_defined_unused`, `-noall_load`, `-no_dead_strip_inits_and_terms`, `-nofixprebinding`, `-nomultidefs`, `-noprebind`, `-noseglinkedit`, `-pagezero_size`, `-prebind`, `-prebind_all_twolevel_modules`, `-private_bundle`, `-read_only_relocs`, `-sectalign`, `-sectobjectsymbols`, `-whyload`, `-seg1addr`, `-sectcreate`, `-sectobjectsymbols`, `-sectorder`, `-segaddr`, `-segs_read_only_addr`, `-segs_read_write_addr`, `-seg_addr_table`, `-seg_addr_table_filename`, `-seglinkedit`, `-segprot`, `-segs_read_only_addr`, `-segs_read_write_addr`, `-single_module`, `-sub_library`, `-sub_umbrella`, `-twolevel_namespace`, `-umbrella`, `-undefined`, `-unexported_symbols_list`, `-weak_reference_mismatches`, `-whatsloaded`, and `-whyload`.

FR-V Options

The Fujitsu FR-V is a Very Long Instruction Word (VLIW) processor core that was developed to support embedded applications in the digital consumer electronics, cell phone, mobile electronics, and automotive navigation markets. The FR-V is a very low-power core that provides a 16-bit integer instruction set, a 32-bit integer instruction set, a media instruction set, a digital signal instruction set, and a floating-point instruction set. The floating-point instruction set is based on the IEEE 754 specification.

GCC options when compiling code for the FR-V processor are the following:

`-macc-4`: Specifying this option causes the GCC compilers to only use the first four media accumulator registers.

`-macc-8`: Specifying this option causes the GCC compilers to use all eight media accumulator registers.

`-malign-labels`: Specifying this option causes the GCC compilers to align labels to an 8-byte boundary by inserting NOOPs. This option only has an effect when VLIW packing is enabled with `-mpack`.

`-malloc-cc`: Specifying this option causes the GCC compilers to dynamically allocate condition code registers.

`-mcond-exec`: Specifying this option causes the GCC compilers to use conditional execution and is active by default. This option is primarily used for debugging the compiler itself and may be removed in future versions of GCC.

`-mcond-move`: Specifying this option causes the GCC compilers to enable the use of conditional-move instructions. This option is active by default. This option is primarily used for debugging the compiler itself and may be removed in future versions of GCC.

`-mcpu=`*`CPU`*: Specifying this option enables you to specify the type of processor for which the GCC compilers will generate code. Possible values for *`CPU`* are `frv`, `fr550`, `tomcat`, `fr500`, `fr450`, `fr405`, `fr400`, `fr300`, and `simple`.

`-mdouble`: Specifying this option causes the GCC compilers to use an ABI that supports double-word floating-point instructions.

`-mdword`: Specifying this option causes the GCC compilers to use an ABI that supports double-word instructions.

`-mfdpic`: Specifying this option causes the GCC compilers to use the FDPIC ABI, which uses function descriptors to represent pointers to functions. If no `PIC`- or `PIE`-related options are specified, using this option implies `-fPIE`. If the `-fpic` or `-fpie` options are specified, using this option assumes that global offset table (GOT) entries and small data are within a 12-bit range from the GOT base address. If the `-fPIC` or `-fPIE` options are specified, GOT offsets are computed with 32 bits.

`-mfixed-cc`: Specifying this option causes the GCC compilers to use only `icc0` and `fcc0` rather than trying to dynamically allocate condition code registers.

`-mfpr-32`: Specifying this option causes the GCC compilers to use only the first 32 floating-point registers.

`-mfpr-64`: Specifying this option causes the GCC compilers to use all 64 floating-point registers.

`-mgpr-32`: Specifying this option causes the GCC compilers to use only the first 32 general-purpose registers.

`-mgpr-64`: Specifying this option causes the GCC compilers to use all 64 general-purpose registers.

`-mgprel-ro`: Specifying this option causes the GCC compilers to enable the use of GPREL (global pointer relative) relocations in the FDPIC ABI for data that is known to be in read-only sections. This option is enabled by default unless the `-fpic` or `-fpie` options are being used. When the `-fPIC` or `-fPIE` options are being used, this option avoids the need for GOT entries for referenced symbols. If you need these entries, you can use this option to enable them.

`-mhard-float`: Specifying this option causes the GCC compilers to use hardware instructions for floating-point operations.

`-minline-plt`: Specifying this option causes the GCC compilers to enable inlining of PLT (procedure linkage table) entries in function calls to functions that are not known to bind locally. This option has no effect unless the `-mfdpic` option is also specified. This option is enabled by default if the `-fPIC` or `-fpic` options are being used, or when optimization options such as `-O3` or above are specified.

`-mlibrary-pic`: Specifying this option causes the GCC compilers to generate position-independent extended application binary interface (EABI) code.

`-mlinked-fp`: Specifying this option causes the GCC compilers to conform to the EABI requirement of always creating a frame pointer whenever a stack frame is allocated. This option is enabled by default and can be disabled with `-mno-linked-fp`.

`-mlong-calls`: Specifying this option causes the GCC compilers to use indirect addressing to call functions outside the current compilation unit, which enables functions to be placed anywhere within the 32-bit address space.

`-mmedia`: Specifying this option enables the GCC compilers to use media instructions.

`-mmuladd`: Specifying this option causes the GCC compilers to use the multiply and add/subtract instructions.

`-mmulti-cond-exec`: Specifying this option causes the GCC compilers to optimize the && and || operations in conditional execution and is active by default. This option is primarily used for debugging the compiler itself and may be removed in future versions of GCC.

`-mnested-cond-exec`: Specifying this option causes the GCC compilers to enable nested conditional execution optimizations and is active by default. This option is primarily used for debugging the compiler itself and may be removed in future versions of GCC.

`-mno-cond-exec`: Specifying this option prevents the GCC compilers from using conditional execution. This option is primarily used for debugging the compiler itself and may be removed in future versions of GCC.

`-mno-cond-move`: Specifying this option prevents the GCC compilers from using conditional-move instructions. This option is primarily used for debugging the compiler itself and may be removed in future versions of GCC.

`-mno-double`: Specifying this option prevents the GCC compilers from using double-word floating-point instructions.

`-mno-dword`: Specifying this option prevents the GCC compilers from using double-word instructions.

`-mno-eflags`: Specifying this option prevents the GCC compilers from marking ABI switches in `e_flags`.

`-mno-media`: Specifying this option prevents the GCC compilers from using media instructions.

`-mno-muladd`: Specifying this option prevents the GCC compilers from using multiply and add/subtract instructions.

`-mno-multi-cond-exec`: Specifying this option prevents the GCC compilers from optimizing the && and || operations in conditional execution. This option is primarily used for debugging the compiler itself and may be removed in future versions of GCC.

`-mno-nested-cond-exec`: Specifying this option prevents the GCC compilers from optimizing nested conditional execution. This option is primarily used for debugging the compiler itself and may be removed in future versions of GCC.

`-mno-optimize-membar`: Specifying this option prevents the GCC compilers from automatically removing redundant membar instructions from the code that they generate.

`-mno-pack`: Specifying this option prevents the GCC compilers from packing VLIW instructions.

`-mno-scc`: Specifying this option prevents the GCC compilers from using conditional set instructions. This option is primarily used for debugging the compiler itself and may be removed in future versions of GCC.

`-mno-vliw-branch`: Specifying this option prevents the GCC compilers from running a pass to pack branches into VLIW instructions. This option is primarily used for debugging the compiler itself and may be removed in future versions of GCC.

`-moptimize-membar`: Specifying this option causes the GCC compilers to remove redundant membar instructions from the code that they generate. This option is enabled by default.

`-mpack`: Specifying this option causes the GCC compilers to pack VLIW instructions. This option is the default.

`-mscc`: Specifying this option causes the GCC compilers to use conditional set instructions and is active by default. This option is primarily used for debugging the compiler itself and may be removed in future versions of GCC.

`-msoft-float`: Specifying this option causes the GCC compilers to use library routines for floating-point operations.

`-mTLS`: Specifying this option causes the GCC compilers to assume a large TLS (thread-local storage) segment when generating thread-local code.

`-mtls`: Specifying this option prevents the GCC compilers from assuming that a large TLS segment is being used when generating thread-local code.

`-mtomcat-stats`: Specifying this option causes the GCC compilers to cause the GNU assembler to print Tomcat processor statistics.

`-multilib-library-pic`: Specifying this option causes the GCC compilers to link with the libraries that support position-independent code, and is implied by the `-mlibrary-pic` option, as well as by `-fPIC` and `-fpic` unless the `-mfdpic` option is specified. You should never have to explicitly supply this option.

`-mvliw-branch`: Specifying this option causes the GCC compilers to run an extra pass to pack branches into VLIW instructions and is active by default. This option is primarily used for debugging the compiler itself and may be removed in future versions of GCC.

H8/300 Options

H8 is a family of 8-bit microprocessors featuring an H8/300 CPU core and a variety of on-chip supporting modules that provide a variety of system functions. Often used as microcontrollers in embedded environments, H8/300-based microprocessors are available from Hitachi, SuperH, and now Renesas.

GCC options available when compiling code for H8/300-based systems are the following:

`-malign-300`: Specifying this option when compiling for the H8/300H and the H8/S processors tell GCC to use the same alignment rules as for the H8/300. The default for the H8/300H and H8/S is to align `long`s and `float`s on 4-byte boundaries. Specifying the `-malign-300` option causes `long`s and `float`s to be aligned on 2-byte boundaries. This option has no effect on the H8/300.

`-mh`: Specifying this option tells GCC to generate code for the H8/300H processor.

`-mint32`: Specifying this option tells GCC to make `int` data 32 bits long by default, rather than 16 bits long.

`-mn`: Specifying this option tells GCC to generate code for the H8S and H8/300 processors in normal mode. Either the `-mh` or the `-ms` switches must also be specified when using this option.

`-mrelax`: Specifying this option tells GCC to shorten some address references at link time, when possible, by passing the `-relax` option to ld.

`-ms`: Specifying this option tells GCC to generate code for the H8/S processor.

`-ms2600`: Specifying this option tells GCC to generate code for the H8/S2600 processor. The `-ms` option must also be specified when using this option.

HP/PA (PA/RISC) Options

HP/PA stands for Hewlett-Packard Precision Architecture, the original name for what are now commonly referred to as Precision Architecture, Reduced Instruction Set Computing (PA-RISC) systems. PA-RISC is a microprocessor architecture developed by Hewlett-Packard's Systems and VLSI Technology Operation and owes some of its design to the RISC technologies introduced in Apollo's DN10000 RISC systems. Apollo was consumed by HP in the late 1980s, a sad time for us all. PA-RISC CPUs are used in many later HP workstations.

GCC options available when compiling code for PA-RISC systems are the following:

`-march=`*`architecture-type`*: Specifying this option tells GCC to generate code for the specified architecture. The choices for *`architecture-type`* are 1.0 for PA 1.0, 1.1 for PA 1.1, and 2.0 for PA 2.0 processors. The file /usr/lib/sched.models on an HP-UX system identifies the proper architecture option for specific machines. Code compiled for lower numbered architectures will run on higher numbered architectures, but not the other way around.

`-mbig-switch`: Specifying this option tells GCC to generate code suitable for big switch tables. You should only use this option if the assembler or linker complains about branches being out of range within a switch table.

`-mdisable-fpregs`: Specifying this option tells GCC to prevent floating-point registers from being used. This option is used when compiling kernels that perform lazy context switching of floating-point registers. GCC will abort compilation if you use this option and attempt to perform floating-point operations in the application that you are compiling.

`-mdisable-indexing`: Specifying this option tells GCC not to use indexing address modes. You should only use this option if you are running GCC on a PA-RISC system running Mach—and what are the chances of that?

`-mfast-indirect-calls`: Specifying this option tells GCC to generate code that assumes calls never cross space boundaries. This enables GCC to generate code that performs faster indirect calls. This option will not work in nested functions or when shared libraries are being used.

`-mhp-ld`: Specifying this option tells GCC to use options specific to the HP linker. For example, this passes the `-b` option to the linker when building shared libraries. This option does not have any effect on the linker itself, just on the options that are passed to it. This option is only available on 64-bit versions of GCC.

`-mgas`: Specifying this option enables GCC to use assembler directives that are only understood by the GNU assembler. This option should therefore not be used if you are using the HP assembler with GCC.

`-mgnu-ld`: Specifying this option tells GCC to use options specific to the GNU linker. For example, this passes the `-shared` option to ld when building a shared library. This option is only available on 64-bit versions of GCC.

`-mjump-in-delay`: Specifying this option tells GCC to fill the delay slots of function calls with unconditional jump instructions by modifying the return pointer for the function call to be the target of the jump.

`-mlinker-opt`: Specifying this option tells GCC to enable the optimization pass in the HP-UX linker. Optimization makes symbolic debugging impossible, but will provide improved performance in most cases. If you are using GCC on HP-UX 8 or 9 systems, using the HP-UX linker may display erroneous error messages when linking some programs.

`-mlong-calls`: Specifying this option causes GCC to use long call sequences in order to ensure that a function call is always able to reach the stubs generated by the linker.

`-mlong-load-store`: Specifying this option tells GCC to generate the three-instruction load and store sequences that are sometimes required by the HP-UX 10 linker. This option should be unnecessary if you are using the standard GNU linker/loader. This option is the same as the `+k` option provided by HP compilers.

`-mno-space-regs`: Specifying this option tells GCC to generate code that assumes the target has no space registers. This option enables GCC to generate faster indirect calls and use unscaled index address modes. Typically, this option should only be used on PA-RISC 1.0 systems or when compiling a kernel.

`-mpa-risc-1-0`: A deprecated synonym for the `-march=1.0` option.

`-mpa-risc-1-1`: A deprecated synonym for the `-march=1.1` option.

`-mpa-risc-2-0`: A deprecated synonym for the `-march=2.0` option.

`-mportable-runtime`: Specifying this option tells GCC to use the portable calling conventions proposed by HP for ELF systems.

`-mschedule=`*`CPU-type`*: Specifying this option tells GCC to schedule code according to the constraints for the machine type `CPU-type`. Possible choices for *`CPU-type`* are `700`, `7100`, `7100LC`, `7200`, and `8000`. The file /usr/lib/sched.models on an HP-UX system shows the proper scheduling option for your machine. The default value is `8000`.

`-msio`: Specifying this option causes GCC to generate the predefined symbol `_SIO` (server IO). See the `-mwsio` option for related predefines.

`-msoft-float`: Specifying this option tells GCC to generate output that contains library calls for floating-point operations. These libraries are not provided as part of GCC, but are normally found on the target system and should be resolved by the C loader on the target machine if they are available. When using this option and cross-compiling, you must provide suitable libraries containing at least function stubs on the host system, or these function calls will be identified as unresolved.

■ **Tip** Specifying the `-msoft-float` option changes the calling conventions in the output file. All of the modules of a program must be compiled using this option in order to be successfully linked. You will also need to compile libgcc.a, the library that comes with GCC, with `-msoft-float` in order to use this option when compiling applications.

`-munix=`*`UNIX-STD`*: Specifying this option causes GCC to generate compiler predefines and select the appropriate start file for the specified *`UNIX-STD`*. Possible values for *`UNIX-STD`* are `93`, `95`, and `98`. The value `93` is supported on all versions of HP-UX, the value `95` is only supported on HP-UX 10.10 and later, and the value `98` is only supported on HP-UX 11.11 and later. The default values for UNIX-STD are `93` on HP-UX 10.0, `95` on HP-UX 10.10 through 11.00, and `98` for HP-UX 11.11 and later releases. Specifying `-munix=93` provides the same predefines as GCC 3.3 and 3.4. Specifying `-munix=95` provides additional predefines for `XOPEN_UNIX` and `_XOPEN_SOURCE_EXTENDED`, and uses the startfile unix95.o. Specifying `-munix=98` provides additional predefines for `_XOPEN_UNIX`, `_XOPEN_SOURCE_EXTENDED`, `_INCLUDE__STDC_A1_SOURCE`, and `_INCLUDE_XOPEN_SOURCE_500`, and uses the startfile unix98.o.

`-mwsio`: Specifying this option causes GCC to generate the predefines `__h9000s700`, `__h9000s700__`, and `_WSIO` (workstation IO). This option is the default and can be used on both HP-IX and HP-UX systems. (This option is not available in the GCC 4.x compilers.)

`-static`: Specifying this standard GCC option evokes slightly different behavior on HP-UX systems than when compiling static binaries for other platforms. On HP-UX systems, specifying this option also passes specific options to the linker to resolve a dependency in the `setlocale()` call on `libdld.sl`, which causes the resulting binary to be dynamic. The `-nolibdld` option can be specified to prevent GCC from passing these options to the linker.

`-threads`: Specifying this option causes GCC to add support for multithreading, using the DCE (Distributed Computing Environment) thread library. Specifying this option sets other options for both the preprocessor and the linker.

i386 and AMD x86-64 Options

The Intel x86 family of 32-bit processors, commonly referred to as *i386 processors*, is the best known and most widely used type of processor found in modern desktop computers. It is easy to argue their merits against other popular processors such as Apple's PPC G4 and G5 processors, but it is impossible to argue with their ubiquity. AMD64 is AMD's family of compatible 64-bit processors.

The options in this section can be used when compiling any i386 or x86-64 code. For the discussion of options that can only be used on AMD x86-64 systems, see the section earlier in this appendix titled "AMD x86-64 Options."

GCC options available when compiling code for all i386 and 64-bit AMD systems are the following:

`-m128bit-long-double`: Specifying this option tells GCC to use 128 bits to store `long doubles`, rather than 12 bytes (96 bits) as specified in the i386 application binary interface. Pentium and newer processors provide higher performance when `long doubles` are aligned to 8- or 16-byte boundaries, though this is impossible to reach when using 12-byte `long doubles` for array access. If you specify this option, any structures and arrays containing `long doubles` will change size, and function calling conventions for functions taking `long double` arguments will also be modified.

`-m32`: Specifying this option tells GCC to generate code for a 32-bit environment.

`-m386`: This option is a deprecated synonym for the `-mtune=i386` option.

`-m3dnow`: Specifying this option enables the use of built-in functions that allow direct access to the 3DNow extensions of the instruction set. (3DNow is a multimedia extension added by AMD to enhance graphics performance.)

`-m486`: This option is a deprecated synonym for the `-mtune=i486` option.

`-m64`: Specifying this option tells GCC to generate code for a 64-bit environment.

`-m96bit-long-double`: Specifying this option tells GCC to set the size of `long doubles` to 96 bits as required by the i386 application binary interface. This is the default.

`-maccumulate-outgoing-args`: Specifying this option tells GCC to compute the maximum amount of space required for outgoing arguments in the function prologue. This is faster on most modern CPUs because of reduced dependencies, improved scheduling, and reduced stack usage when the preferred stack boundary is not equal to 2. The downside is a notable increase in code size. Specifying this option implies the `-mno-push-args` option.

`-malign-double`: Specifying this option tells GCC to align `double`, `long double`, and `long long` variables on a two-word boundary. This produces code that runs somewhat faster on Pentium or better systems at the expense of using more memory. Specifying this option aligns structures in a way that does not conform to the published application binary interface specifications for the 80386 processors.

`-march=`*`CPU-type`*: Specifying this option tells GCC to generate instructions for the specified machine type `CPU-type`. Possible values for *`CPU-type`* are `athlon`, `athlon64`, `athlon-4`, `athlon-fx`, `athlon-tbird`, `athlon-mp`, `athlon-xp`, `c3`, `c3-2`, `i386`, `i486`, `i586` (equivalent to `pentium`), `i686` (equivalent to `pentiumpro`), `k6`, `k6-2`, `k6-3`, `k8`, `opteron`, `pentium`, `pentium-m`, `pentium-mmx`, `pentiumpro`, `pentium2`, `pentium3`, `pentium3m`, `pentium4`, `pentium4m`, `prescott`, `winchip-c6`, and `winchip2`. Specifying the `-march=`*`CPU-type`* option implies the `-mtune=`*`CPU-type`* option. See the later discussion of the `-mtune=`*`CPU-type`* option for a verbose explanation of all supported values for *`CPU-type`*.

`-masm=`*`dialect`*: Specifying this option tells GCC to output asm instructions using the selected *dialect*, which can be either `intel` or `att`. The `att` dialect is the default.

`-mcmodel=`*`model`*: Specifying this option causes GCC to generate code for the indicated memory model. Valid values for *model* are `small` (the program and all symbols must be linked in the lower 2GB of address space and use 64-bit pointers), `kernel` (for programs such as the Linux kernel, which run in the negative 2GB of the address space), `medium` (the program must be linked in the lower 2GB of address space but symbols can be anywhere in the address space—this model can not be used to build shared libraries), and `large` (anything can be anywhere in the address space, though GCC does not currently support this code model).

`-mcpu=`*`CPU-type`*: This option is a deprecated synonym for the `-mtune=`*`CPU-type`* option.

`-mfpmath=`*`unit`*: Specifying this option tells GCC to generate floating-point arithmetic for the floating-point unit *`unit`*. Valid choices for *`unit`* are the following:

- `387`: Use the standard 387 floating-point coprocessor present in most i386 chips and emulated otherwise. Code compiled with this option will run almost everywhere. The temporary results are computed in 80-bit precision instead of precision specified by the type, resulting in slightly different results compared to most other chips. See the standard GCC option `-ffloat-store` for more information. This is the default if the `-mfpmath` option is not specified.

- `sse`: Use scalar floating-point instructions present in the Streaming SIMD Extensions (SSE) instruction set. This instruction set is supported by Intel Pentium III and newer chips, and in the Athlon 4, Athlon XP, and Athlon MP chips from AMD. Earlier versions of the SSE instruction set only supported single-precision arithmetic, which means that double- and extended-precision math is still done using the standard 387 instruction set. The versions of the SSE instruction set provided in Pentium 4 and AMD x86-64 chips provide direct support for double-precision arithmetic. If you are compiling for chips other than these, you should use the `-march=`*`CPU-type`*, `-msse`, and `-msse2` options to enable SSE extensions and to use this option effectively. The `-mfpmath=sse` option is the default for GCC compiled for the x86-64 target.
- `sse,387`: Attempt to utilize both instruction sets at once. This effectively doubles the number of available registers, as well as the amount of execution resources on chips with separate execution units for 387 and SSE. Using this option may cause problems because the GCC register allocator does not always model separate functional units well.

`-mieee-fp`: Specifying this option tells GCC to use IEEE floating-point comparisons, which correctly handle the case where the result of a comparison is unordered.

`-minline-all-stringops`: Specifying this option tells GCC to inline all string operations. By default, GCC only inlines string operations when the destination is known to be aligned to at least a 4-byte boundary. This enables more inlining, which increases code size, but may also improve the performance of code that depends on fast `memcpy()`, `strlen()`, and `memset()` for short lengths.

`-mlarge-data-threshold=`*`number`*: Specifying this option when `-mcmodel=medium` is used tells GCC that data greater than *`number`* should be placed in a large data section. The value for *`number`* must be the same across all of the object code that you are linking together and defaults to `65535`.

`-mmmx`: Specifying this option enables the use of built-in functions that allow direct access to the MMX extensions of the instruction set.

`-mno-3dnow`: Specifying this option disables the use of built-in functions that allow direct access to the 3DNow extensions of the instruction set.

`-mno-align-double`: Specifying this option tells GCC not to align `double`, `long double`, and `long long` variables on a two-word boundary, using a one-word boundary instead. This reduces memory consumption but may result in slightly slower code. Specifying this option aligns structures containing these datatypes in a way that conforms to the published application binary interface specifications for the 80386 processor.

`-mno-align-stringops`: Specifying this option tells GCC not to align the destination of inlined string operations. This switch reduces code size and improves performance when the destination is already aligned.

`-mno-fancy-math-387`: Specifying this option tells GCC to avoid generating the sin, cos, and sqrt instructions for the 80387 floating-point units because these instructions are not provided by all 80387 emulators. This option has no effect unless you also specify the `-funsafe-math-optimizations` option. This option is overridden when `-march` indicates that the target CPU will always have an FPU and so the instruction will not need emulation. This option is the default for GCC on FreeBSD, OpenBSD, and NetBSD systems.

`-mno-fp-ret-in-387`: Specifying this option tells GCC not to use the FPU registers for function return values, returning them in ordinary CPU registers instead. The usual calling convention on 80386 systems returns `float` and `double` function values in an FPU register, even if there is no FPU. This assumes that the operating system provides emulation support for an FPU, which may not always be correct.

`-mno-ieee-fp`: Specifying this option tells GCC not to use IEEE floating-point comparisons.

`-mno-mmx`: Specifying this option disables the use of built-in functions that allow direct access to the MMX extensions of the instruction set.

`-mno-push-args`: Specifying this option tells GCC to use standard `SUB/MOV` operations to store outgoing parameters, rather than the potentially smaller `PUSH` operation. In some cases, using `SUB/MOV` rather than `PUSH` may improve performance because of improved scheduling and reduced dependencies.

`-mno-red-zone`: Specifying this option tells GCC not to use the red zone mandated by the x86-64 ABI. The red zone is a 128-byte area beyond the stack pointer that will not be modified by signal or interrupt handles, and can therefore be used for temporary data storage without adjusting the stack pointer.

`-mno-sse`: Specifying this option disables the use of built-in functions that allow direct access to the SSE extensions of the instruction set.

`-mno-sse2`: Specifying this option disables the use of built-in functions that allow direct access to the SSE2 extensions of the instruction set.

`-mno-svr3-shlib`: Specifying this option tells GCC to place uninitialized local variables in the data segment. This option is only meaningful on System V Release 3 (SVR3) systems.

`-mno-tls-direct-seg-refs`: Specifying this option enables GCC to add the thread base pointer when accessing thread-local storage variables through offsets from the TLS segment. This is necessary if the operating system that you are compiling for does not map the TLS segment to cover the entire TLS area.

`-momit-leaf-frame-pointer`: Specifying this option tells GCC not to keep the frame pointer in a register for leaf functions. This avoids generating the instructions to save, set up, and restore frame pointers and makes an extra register available in leaf functions. The option `-fomit-frame-pointer` removes the frame pointer for all functions, which might make debugging harder.

`-mpentium`: This option is a deprecated synonym for the `-mtune=i586` and `-mtune=pentium` options.

`-mpentiumpro`: This option is a deprecated synonym for the `-mtune=i686` and `-mtune=pentiumpro` options.

`-mpreferred-stack-boundary=`*num*: Specifying this option tells GCC to attempt to keep the stack boundary aligned to a 2**num* byte boundary. This extra alignment consumes extra stack space and generally increases code size. Code that is sensitive to stack space usage, such as embedded systems and operating system kernels, may want to reduce the preferred alignment to `-mpreferred-stack-boundary=2`. If the `-mpreferred-stack-boundary` option is not specified, the default is 4 (16 bytes or 128 bits), except when optimizing for code size (using the `-Os` option), in which case the default is the minimum correct alignment (4 bytes for x86, and 8 bytes for x86-64).

On Pentium and Pentium Pro systems, `double` and `long double` values should be aligned to an 8-byte boundary (see the `-malign-double` option), or they will incur significant runtime performance penalties. On Pentium III systems, the SSE datatype `__m128` incurs similar penalties if it is not 16-byte aligned.

■ **Note** To ensure proper alignment of the values on the stack, the stack boundary must be as aligned as required by any value stored on the stack. Therefore, every function and library must be generated such that it keeps the stack aligned. Calling a function compiled with a higher preferred stack boundary from a function compiled with a lower preferred stack boundary will probably misalign the stack. The GNU folks recommend that libraries that use callbacks always use the default setting.

`-mpush-args`: Specifying this option tells GCC to use the `PUSH` operation to store outgoing parameters. This method is shorter and usually at least as fast as using `SUB/MOV` operations. This is the default.

`-mregparm=`*num*: Specifying this option controls the number of registers used to pass integer arguments, specifid as *num*. By default, no registers are used to pass arguments, and at most, three registers can be used. You can also control this behavior for a specific function by using the function attribute `regparm`.

■ **Note** If you use this switch to specify a nonzero number of registers, you must build all modules with the same value, including any libraries that the program uses. This includes system libraries and startup modules.

`-mrtd`: Specifying this option tells GCC to use a function-calling convention where functions that take a fixed number of arguments return with the `ret(`*num*`)` instruction, which pops their arguments during the return. This saves one instruction in the caller because there is no need to pop the arguments there. This calling convention is incompatible with the one normally used on Unix, and therefore cannot be used if you need to call libraries that have been compiled with generic Unix compilers. When using this option, you must provide function prototypes for all functions that take variable numbers of arguments (including `printf()`); if you do not, incorrect code will be generated for calls to those functions. You must also be careful to never call functions with extra arguments, which would result in seriously incorrect code. This option takes its name from the 680x0 rtd instruction.

■ **Tip** To optimize heavily used functions, you can specify that an individual function is called with the calling sequence indicated by the `-mrtd option` by using the function attribute `stdcall`. You can also override the `-mrtd` option for specific functions by using the function attribute `cdecl`.

`-msoft-float`: Specifying this option tells GCC to generate output that contains library calls for floating-point operations. These libraries are not provided as part of GCC but are normally found on the target system and should be resolved by the C loader on the target machine if they are available. When using this option and cross-compiling, you must provide suitable libraries containing at least function stubs on the host system, or these function calls will be identified as unresolved.

■ **Note** On machines where a function returns floating-point results in the 80387 register stack, some floating-point opcodes may be emitted even if `-msoft-float` is used.

`-msse`: Specifying this option enables the use of built-in functions that allow direct access to the SSE extensions of the instruction set.

`-msse2`: Specifying this option enables the use of built-in functions that allow direct access to the SSE2 extensions of the instruction set.

`-msselibm`: Specifying this option causes GCC to use special versions of certain math library (`libm`) options that come with an SSE ABI implementation. This option is useful with the `-mfpmath=sse` option to avoid moving values between SSE registers and the i387 floating-point stack.

`-msseregparm`: Specifying this option causes GCC to use the SSE register passing conventions for floating-point and `double` arguments and return values. If you use this option, all object code and libraries that will be linked or used together must also be compiled with this option.

`-msvr3-shlib`: Specifying this option tells GCC to place uninitialized local variables in the bss segment. This option is only meaningful on System V Release 3 (SVR3) systems.

`-mthreads`: Specifying this option tells GCC to support thread-safe exception handling on Mingw32 platforms. Code that relies on thread-safe exception handling must compile and link all code with this option. When compiling, specifying the `-mthreads` option sets the `-D_MT` symbol. When linking, specifying this option links in a special thread helper library (equivalent to the `-lmingwthrd` option) that cleans up per-thread exception-handling data.

`-mtls-direct-seg-refs`: Specifying this option enables GCC to access thread-local storage variables through offsets from the TLS segment, without adding the thread base pointer. Whether or not this is valid depends on whether the operating system that you are compiling for maps the TLS segment to cover the entire TLS area. This option is active by default when using Glibc.

`-mtune=`*`CPU-type`*: Specifying this option tells GCC to generate object code that is tuned to and scheduled for the specified *`CPU-type`*, with the exception of the ABI and the set of available instructions. (The instruction set is selected using the `-march=`*`CPU-type`* option.) Possible values for *`CPU-type`* are the following (organized alphabetically):

- `athlon`, `athlon-tbird`: AMD Athlon CPUs with MMX, 3DNow, enhanced 3DNow, and SSE prefetch instruction set support.
- `athlon-4`, `athlon-xp`, `athlon-mp`: Improved AMD Athlon CPUs with MMX, 3DNow, enhanced 3DNow, and full SSE instruction set support.
- `c3`: VIA C3 CPU with MMX and 3DNow instruction set support.
- `c3-2`: VIA C3-2 CPU with MMX and SSE instruction set support.
- `generic`: IA32/AMD64/EM64T processors. You should only use this option if you do not know the specific `CPU-type` on which people will be running your code. Note that there is no corresponding `-march=generic` option, because the `-march` option identifies the instruction set the compiler can use, and no generic instruction set is applicable to all processors.
- `i386`: Original Intel i386 CPU.
- `i486`: Intel's i486 CPU.
- `i586`, `pentium`: Intel Pentium CPU without MMX support.
- `i686`: All processors in the i686 family. The same as `generic`, except that when this option is specified with `-march=i686`, the resulting code will use the Pentium Pro instruction set.
- `k6`: AMD K6 CPU with MMX instruction set support.

- `k6-2`, `k6-3`: Improved versions of the AMD K6 CPU with MMX and 3DNow instruction set support.
- `k8`, `opteron`, `athlon64`, `athlon-fx`: CPUs based on the AMD K8 core with x86-64 instruction set support, which provides a superset of the MMX, SSE, SSE2, 3DNow, enhanced 3DNow, and 64-bit instruction set extensions.
- `nocona`: Improved version of the Intel Pentium 4 CPU with 64-bit extensions, MMX, SSE, SSE2, and SSE3 instruction set support.
- `pentium-m`: Low-power version of the Intel Pentium 3 CPU with MMX, SSE, and SSE2 instruction set support.
- `pentium-mmx`: Intel Pentium core CPU with MMX instruction set support.
- `pentiumpro`: Intel Pentium Pro CPU.
- `pentium2`: Intel Pentium 2 CPU based on the Pentium Pro core, but with MMX instruction set support.
- `pentium3`, `pentium3m`: Intel Pentium 3 CPUs based on the Pentium Pro core, but with MMX and SSE instruction set support.
- `pentium4`, `pentium4m`: Intel Pentium 4 CPUs with MMX, SSE, and SSE2 instruction set support.
- `prescott`: Improved version of the Intel Pentium 4 CPU with MMX, SSE, SSE2, and SSE3 instruction set support.
- `winchip-c6`: IDT WinChip C6 CPU, which is essentially similar to i486 with additional MMX instruction set support.
- `winchip2`: IDT WinChip 2 CPU, which is essentially similar to i486 with additional MMX and 3DNow instruction set support.

■ **Note** Picking a specific *CPU-type* using the `-mtune=`*CPU-type* option will schedule the code appropriately for that particular chip, but GCC will not generate any code that does not run on the i386 without the `-march=`*CPU-type* option being used to identify a particular *CPU-type*.

IA-64 Options

The options in this section can only be used when compiling code targeted for 64-bit Intel processors (the GCC ia-64 build target). This does not include EMT64 chips, which are supported as a subset of the x86-64 build target. For a discussion of options that can be used on i386 systems, see the section earlier in this appendix titled "i386 and AMD x86-64 Options." For a discussion of options that can only be used on AMD64 (64-bit AMD) systems, see the section earlier in this appendix titled "AMD x86-64 Options."

GCC options available when compiling code for 64-bit Intel systems are the following:

`-mauto-pic`: Specifying this option tells GCC to generate position-independent code (i.e., code that is self-relocatable). Specifying this option implies the `-mconstant-gp` option. This option is useful when compiling firmware code.

`-mb-step`: Specifying this option tells GCC to generate code that works around Itanium B step errata.

`-mbig-endian`: Specifying this option tells GCC to generate code for a big endian target. This is the default for HP-UX systems using the IA-64 processor.

`-mconstant-gp`: Specifying this option tells GCC to generate code that uses a single constant global pointer value. This is useful when compiling kernel code because all pointers are offset from a single global.

`-mdwarf2-asm`: Specifying this option tells GCC to generate assembler code for DWARF2 line number debugging. This information may be useful when not using the GNU assembler.

`-mearly-stop-bits`: Specifying this option enables GCC to put stop bits in object code earlier than immediately preceding the instruction that generated the stop bits. This may help with scheduling.

`-mfixed-range=`*`register-range`*: Specifying this option tells GCC to generate code that treats the specified range of registers as fixed registers. A *register range* is specified as two registers separated by a dash. Multiple register ranges can be specified separated by a comma. A fixed register is one that the register allocator cannot use. This option is useful when compiling kernel code.

`-mgnu-as`: Specifying this option tells GCC to generate code for the GNU assembler. This is the default.

`-mgnu-ld`: Specifying this option tells GCC to generate code for the GNU linker. This is the default.

`-milp32`: Specifying this option causes GCC to generate code for a 32-bit environment. This option is specific to the HP-UX platform.

`-milp64`: Specifying this option causes GCC to generate code for a 64-bit environment. This option is specific to the HP-UX platform.

`-minline-float-divide-max-throughput`: Specifying this option tells GCC to generate code for inline floating-point division using a maximum throughput algorithm.

`-minline-float-divide-min-latency`: Specifying this option tells GCC to generate code for inline floating-point division using a minimum latency algorithm.

`-minline-int-divide-max-throughput`: Specifying this option tells GCC to generate code for inline integer division using a maximum throughput algorithm.

`-minline-int-divide-min-latency`: Specifying this option tells GCC to generate code for inline integer division using a minimum latency algorithm.

`-minline-sqrt-max-throughput`: Specifying this option tells GCC to generate inline code for square root calculations using a maximum throughput algorithm.

`-minline-sqrt-min-latency`: Specifying this option tells GCC to generate inline code for square root calculations using a minimum latency algorithm.

`-mlittle-endian`: Specifying this option tells GCC to generate code for a little endian target. This is the default for AIX 5 and Linux systems using the IA-64 processor.

`-mno-dwarf2-asm`: Specifying this option tells GCC not to generate assembler code for DWARF2 line number debugging.

`-mno-early-stop-bits`: Specifying this option prevents GCC from putting stop bits in object code anywhere other than immediately preceding the instruction that generated the stop bits.

`-mno-gnu-as`: Specifying this option tells GCC not to generate code for the GNU assembler, but to assume that assembly will be done using a system assembler.

`-mno-gnu-ld`: Specifying this option tells GCC not to generate code for the GNU linker, but to assume linking/loading will be done using a system linker.

`-mno-inline-float-divide`: Specifying this option prevents GCC from generating code for inline floating point division.

`-mno-inline-int-divide`: Specifying this option prevents GCC from generating code for inline integer division.

`-mno-pic`: Specifying this option tells GCC to generate code that does not use a global pointer register. This results in code that is not position-independent and violates the IA-64 ABI.

`-mno-register-names`: Specifying this option tells GCC not to generate `in`, `loc`, and `out` register names for the stacked registers.

`-mno-sched-ar-data-spec`: Specifying this option tells GCC to disable data speculative scheduling after reload using the ld.a and associated chk.a instructions.

`-mno-sched-ar-in-data-spec`: Specifying this option tells GCC to disable speculative scheduling of the instructions that are dependent on data-speculative scheduling loads after reload.

`-mno-sched-br-data-spec`: Specifying this option tells GCC to disable data-speculative scheduling before reload using the ld.a and associated chk.a instructions.

`-mno-sched-br-in-data-spec`: Specifying this option tells GCC to disable speculative scheduling of the instructions that are dependent on data-speculative scheduling loads before reload.

`-mno-sched-contol-ldc`: Specifying this option tells GCC to disable the use of the ld.c instruction to check control-speculative loads.

`-mno-sched-control-spec`: Specifying this option tells GCC to disable control-speculative scheduling using the ld.s and associated chk.s instructions.

`-mno-sched-count-spec-in-critical-path`: Specifying this option tells GCC not to consider speculative dependencies when computing instruction priorities.

`-mno-sched-in-control-spec`: Specifying this option tells GCC to disable speculative scheduling of instructions that are dependent on control-speculative scheduling of load instructions.

`-mno-sched-ldc`: Specifying this option tells GCC to disable simple data speculation checks using the ld.c instruction and to only use the chk.c instruction.

`-mno-sched-prefer-non-control-spec-insns`: Specifying this option enables GCC to use control-speculative scheduling instructions whenever possible.

`-msched-prefer-non-data-spec-insns`: Specifying this option enables GCC to use data-speculative scheduling instructions whenever possible.

`-mno-sched-spec-verbose`: Specifying this option tells GCC not to display information about speculative scheduling.

`-mno-sdata`: Specifying this option tells GCC to disable optimizations that use the small data section. This option may be useful for working around optimizer bugs.

`-mno-volatile-asm-stop`: Specifying this option tells GCC not to generate a stop bit immediately before and after volatile `asm` statements.

`-mt`: Specifying this option sets flags for the preprocessor and linker that add support for multi-threading using the POSIX threading library. This option is specific to the HP-UX platform.

`-mregister-names`: Specifying this option tells GCC to generate `in`, `loc`, and `out` register names for the stacked registers. This may make assembler output more readable.

`-msched-ar-data-spec`: Specifying this option tells GCC to use data-speculative scheduling after reload using the ld.a and associated chk.a instructions. This option is enabled by default.

`-msched-ar-in-data-spec`: Specifying this option tells GCC to perform speculative scheduling of the instructions that are dependent on data-speculative scheduling load after reload, and is only useful if `-msched-ar-data-spec` is also specified. This option is enabled by default.

`-msched-br-data-spec`: Specifying this option tells GCC to use data-speculative scheduling before reload using the ld.a and associated chk.a instructions. This option is disabled by default.

`-msched-br-in-data-spec`: Specifying this option tells GCC to perform speculative scheduling of the instructions that are dependent on data-speculative scheduling load before reload, and is only useful if `-msched-br-data-spec` is also specified. This option is enabled by default.

`-msched-contol-ldc`: Specifying this option tells GCC to use the ld.c instruction to check control-speculative loads. If no speculatively scheduled dependent instructions follow the load, ld.sa is used instead, and ld.c is used to check it. This option is disabled by default.

`-msched-control-spec`: Specifying this option tells GCC to enable control-speculative scheduling of loads using the ld.s and associated chk.s instructions. This option is disabled by default.

`-msched-count-spec-in-critical-path`: Specifying this option tells GCC to consider speculative dependencies when computing instruction priorities. This option is disabled by default.

`-msched-in-control-spec`: Specifying this option tells GCC to enable speculative scheduling instructions that are dependent on control-speculative loads, and is only useful if `-msched-control-spec` is also specified. This option is enabled by default.

`-msched-ldc`: Specifying this option tells GCC to enable simple data-speculation checks using the ld.c instruction. This option is enabled by default.

`-msched-prefer-non-control-spec-insns`: Specifying this option tells GCC to only use control-speculative scheduling instructions when no other choices exist. This causes control speculation to be much more conservative. This option is disabled by default.

`-msched-prefer-non-data-spec-insns`: Specifying this option tells GCC to only use data-speculative scheduling instructions when no other choices exist. This causes data speculation to be much more conservative. This option is disabled by default.

`-msched-spec-verbose`: Specifying this option tells GCC to display information about speculative scheduling.

`-msdata`: Specifying this option tells GCC to enable optimizations that use the small data section. Though this option provides performance improvements, problems have been reported when using these optimizations.

`-mtls-size=`*`tls-tls`*: Specifying this option tells GCC to set the size of immediate TLS in bits to *`tls-size`*. Valid values are `14`, `22`, and `64`.

`-mtune=`*`CPU-type`*: Specifying this option tells GCC to generate object code that is tuned to and scheduled for the specified `CPU-type`. Possible values for *`CPU-type`* are the following (organized alphabetically):

- `itanium`: Intel's original Itanium IA-64 processor, with a clock speed of 733MHz, a ten-stage execution pipeline, and up to 4MB of L3 cache
- `itanium1`: a synonym for the Itanium
- `itanium2`: Intel's second-generation Itanium process, with clock speeds of 900Mhz to 1.7GHz, a shorter, seven-stage execution pipeline, and up to 9MB of L3 cache
- `mckinley`: a synonym for the Itanium 2
- `merced`: a synonym for the Itanium

`-mvolatile-asm-stop`: Specifying this option tells GCC to generate a stop bit immediately before and after volatile `asm` statements.

`-pthread`: Specifying this option sets flags for the preprocessor and linker that add support for multithreading using the POSIX threading library. This option is specific to the HP-UX platform.

Intel 960 Options

Intel's i960 family consists of high-performance, 32-bit embedded RISC processors supported by an outstanding selection of development tools (such as the one that you're reading about). Intel's i960 processors were often used in high-performance, embedded networking and imaging scenarios.

■ **Note** Support for this processor family is not provided in GCC version 4.0. These options are therefore only of interest if you are using a version of GCC that is earlier than 4.x and that you are certain provides support for this processor family.

GCC options available when compiling code for Intel 960 systems are the following:

`-masm-compat` | `-mintel-asm`: Specifying either of these options tells GCC to enable compatibility with the iC960 assembler.

`-mcode-align`: Specifying this option tells GCC to align code to 8-byte boundaries in order to support faster fetching. This option is the default for C-series processors (as specified using the `-mcpu-CPU-type` option).

`-mcomplex-addr`: Specifying this option tells GCC to use a complex addressing mode to provide performance improvements. Complex addressing modes may not be worthwhile on the K-series processors, but they definitely are on the C-series processors. This option is the default for all processors (as specified using the `-mcpu=CPU-type` option) except for the CB and CC processors.

`-mcpu=CPU-type`: Specifying this option tells GCC to use the defaults for the machine type `CPU-type`, affecting instruction scheduling, floating-point support, and addressing modes. Possible values for `CPU-type` are `ka`, `kb`, `mc`, `ca`, `cf`, `sa`, and `sb`, which are different generations and versions of the i960 processor. The default is `kb`.

`-mic-compat`: Specifying this option tells GCC to enable compatibility with the iC960 version 2.0 or version 3.0 C compiler from Intel.

`-mic2.0-compat`: Specifying this option tells GCC to enable compatibility with the iC960 version 2.0 C compiler from Intel.

`-mic3.0-compat`: Specifying this option tells GCC to enable compatibility with the iC960 version 3.0 C compiler from Intel.

`-mleaf-procedures`: Specifying this option tells GCC to attempt to alter leaf procedures to be callable with the `bal` instruction as well as call. This will result in more efficient code for explicit calls when the bal instruction can be substituted by the assembler or linker, but less efficient code in other cases, such as calls via function pointers or when using a linker that does not support this optimization.

`-mlong-double-64`: Specifying this option tells GCC to implement the `long double` type as 64-bit floating-point numbers. If you do not specify this option, `long double`s are implemented by 80-bit floating-point numbers. This option is present because there is currently no 128-bit `long double` support. This option should only be used when using the `-msoft-float` option, for example, for soft-float targets.

`-mno-code-align`: Specifying this option tells GCC not to align code to 8-byte boundaries for faster fetching. This option is the default for all non–C-series implementations (as specified using the `-mcpu-CPU-type` option).

`-mno-complex-addr`: Specifying this option tells GCC not to assume that the use of a complex addressing mode is a win on this implementation of the i960. Complex addressing modes may not be worthwhile on the K-series processors. This option is the default for the CB and CC processors (as specified using the `-mcpu-CPU-type` option).

`-mno-leaf-procedures`: Specifying this option tells GCC to always call leaf procedures with the call instruction.

`-mno-strict-align`: Specifying this option tells GCC to permit unaligned accesses.

`-mno-tail-call`: Specifying this option tells GCC not to make additional attempts (beyond those of the machine-independent portions of the compiler) to optimize tail-recursive calls into branches. This is the default.

`-mnumerics`: Specifying this option tells GCC that the processor supports floating-point instructions.

`-mold-align`: Specifying this option tells GCC to enable structure-alignment compatibility with Intel's GCC release version 1.3 (based on GCC 1.37). It would be really sad if this option was still getting much use. This option implies the `-mstrict-align` option.

`-msoft-float`: Specifying this option tells GCC that floating-point support should not be assumed, and to generate output that contains library calls for floating-point operations. These libraries are not provided as part of GCC but are normally found on the target system and should be resolved by the C loader on the target machine if they are available. When using this option and cross-compiling, you must provide suitable libraries containing at least function stubs on the host system, or these function calls will be identified as unresolved.

`-mstrict-align`: Specifying this option tells GCC not to permit unaligned accesses.

`-mtail-call`: Specifying this option tells GCC to make additional attempts (beyond those of the machine-independent portions of the compiler) to optimize tail-recursive calls into branches. You may not want to do this because the detection of cases where this is not valid is not totally complete.

M32C Options

Renesas M32C processor family is a family of 32-bit RISC microprocessors designed for embedded systems. M32Cs are typically used in industrial, automotive, and communications systems.

GCC options available when compiling code for M32C systems are the following:

`-mcpu=`*name*: Specifying this option identifies the CPU for which GCC should generate code. Possible values for *name* are `r8c` for the R8C/Tiny series, `m16c` for the M16C (up to /60) series, `m32cm` for the M16C/80 series, and `m32c` for the M32C/80 series.

`-msim`: Specifying this option tells GCC that the code that it generates will be run on the simulator. This causes GCC to link in an alternate runtime library that supports capabilities such as file I/O. Do not use this option when generating programs that will run on real hardware; you must provide your own runtime library for any I/O functions that you need.

`-memregs=`*number*: Specifying this option tells GCC the number of memory-based pseudoregisters that it can use in generated code. This allows you to use more registers than are physically available but trades off the costs of juggling physical registers against the costs of using memory instead of registers. All of the object modules that you link together must be compiled with the same value for this option, which means that you must not use this option with the standard GCC runtime libraries unless you compiled them with the same option.

M32R Options

The Renesas M32R processor family is a family of 32-bit RISC microprocessors designed for embedded systems. M32Rs are designed for general industrial and car-mounted systems, digital AV equipment, digital imaging equipment, and portable consumer products.

GCC options available when compiling code for M32R systems are the following:

`-G num`: Specifying this option tells GCC to put global and static objects less than or equal to `num` bytes into the small data or bss sections instead of the normal data or bss sections. The default value of `num` is 8. The `-msdata` option must be set to either `sdata` or `use` for this option to have any effect.

■ **Note** All modules should be compiled with the same `-G num` value. Compiling with different values of `num` may or may not work. If it does not, the linker will display an error message and exit.

`-m32r`: Specifying this option tells GCC to generate code for the generic M32R. This is the default.

`-m32r2`: Specifying this option tells GCC to generate code for the M32R/2 processor.

`-m32rx`: Specifying this option tells GCC to generate code for the M32R/X processor.

`-malign-loops`: Specifying this option causes GCC to align all loops to a 32-byte boundary. This option is disabled by default.

`-mbranch-cost=`*number*: Specifying this option tells GCC how to value branches when generating code. If *number* is 1, branches will be preferred over conditional code. If *number* is 2, conditional code will be preferred over branches.

`-mcode-model=large`: Specifying this option tells GCC to assume that objects may be anywhere in the 32-bit address space (GCC generates seth/add3 instructions to load the addresses of those objects), and to assume that subroutines may not be reachable with the bl instruction (GCC generates the much slower seth/add3/jl instruction sequence).

Tip The addressability of a particular object can be set with the `model` attribute.

`-mcode-model=medium`: Specifying this option tells GCC to assume that objects may be anywhere in the 32-bit address space (GCC generates seth/add3 instructions to load their addresses), and to assume that all subroutines are reachable with the bl instruction.

`-mcode-model=small`: Specifying this option tells GCC to assume that all objects live in the lower 16MB of memory (so that their addresses can be loaded with the ld24 instruction), and to assume that all subroutines are reachable with the bl instruction. This is the default.

`-mdebug`: Specifying this option causes GCC to display diagnostic information about the M32R-specific portions of the compiler.

`-mflush-func=`*`name`*: Specifying this option tells GCC the name of the operating system function to call in order to flush the cache. The default is `__flush_cache_`. This option is only meaningful if the `-mno-flush-trap` option has been specified.

`-mflush-trap=`*`number`*: Specifying this option tells GCC which trap number to use in order to flush the cache. The default is 12. Possible values for *number* are 0 through 15, inclusive.

`-missue-rate=`*`number`*: Specifying this option causes GCC to produce code that issues *number* instructions per cycle. Possible values for *number* are 1 or 2.

`-mno-align-loops`: Specifying this option prevents GCC from aligning all loops of 32-byte boundaries.

`-mno-flush-func`: Specifying this option tells GCC that there is no operating system function that can be used to flush the cache.

`-mno-flush-trap`: Specifying this option tells GCC that the cache cannot be flushed by using a trap.

`-msdata=none`: Specifying this option tells GCC to disable the use of the small data area. The small data area consists of the sections sdata and sbss. Variables will be put into one of the data, bss, or rodata sections (unless the `section` attribute has been specified). This is the default.

Tip Objects may be explicitly put in the small data area with the `section` attribute.

`-msdata=sdata`: Specifying this option tells GCC to put small global and static data in the small data area, but not generate special code to reference them.

`-msdata=use`: Specifying this option tells GCC to put small global and static data in the small data area, and to generate special instructions to reference them.

M680x0 Options

The options in this section are used when employing GCC to compile applications for use on systems that rely on the Motorola 68000 family of processors. These are the 68000, 68010, 68020, 68030, 68040, and 68060 processors. These legendary processors were used in almost all early computer workstations before the advent of RISC processors and are still frequently used as microcontrollers.

GCC options available when compiling code for M680x0 systems are the following:

`-m5200`: Specifying this option tells GCC to generate code for a 520X ColdFire family CPU. This is the default when the compiler is configured for 520X-based systems. You should use this option when compiling code for microcontrollers with a 5200 core, including the MCF5202, MCF5203, and MCF5204 processors. The `-m5200` option implies the `-mnobitfield` option.

`-m68000` | `-mc68000`: Specifying either of these options tells GCC to generate code for a 68000 processor. These are the default when the compiler is configured for 68000-based systems. You should use these options when compiling for microcontrollers with a 68000 or EC000 core, including the 68008, 68302, 68306, 68307, 68322, 68328, and 68356 processors. The `-m68000` option implies the `-mnobitfield` option.

`-m68020` | `-mc68020`: Specifying either of these options tells GCC to generate code for a 68020 processor. This is the default when the compiler is configured for 68020-based systems. Specifying these options also sets the `-mbitfield` option.

`-m68020-40`: Specifying this option tells GCC to generate code for a 68040 processor, without using any instructions introduced since the 68020. This results in code that can run relatively efficiently on 68020/68881, 68030, or 68040 systems. The generated code uses the 68881 instructions that are emulated on the 68040 processor.

`-m68020-60`: Specifying this option tells GCC to generate code for a 68060 processor, without using any instructions introduced since the 68020 processor. This results in code that can run relatively efficiently on 68020/68881, 68030, 68040, or 68060 systems. The generated code uses the 68881 instructions that are emulated on the 68060 processor.

`-m68030`: Specifying this option tells GCC to generate code for a 68030 processor. This is the default when the compiler is configured for 68030-based systems.

`-m68040`: Specifying this option tells GCC to generate code for a 68040 processor. This is the default when the compiler is configured for 68040-based systems. Specifying this option inhibits the use of 68881/68882 instructions that have to be emulated by software on the 68040 processor. You should use this option if your 68040 system does not have code to emulate those instructions.

`-m68060`: Specifying this option tells GCC to generate code for a 68060 processor. This is the default when the compiler is configured for 68060-based systems. This option inhibits the use of 68020 and 68881/68882 instructions that have to be emulated by software on the 68060 processor. You should use this option if your 68060 system does not have code to emulate those instructions.

`-m68881`: Specifying this option tells GCC to generate code containing 68881 instructions for floating point. This is the default for most 68020 systems unless the `--nfp` option was specified when GCC was configured.

`-malign-int`: Specifying this option tells GCC to align `int`, `long`, `long long`, `float`, `double`, and `long double` variables on a 32-bit boundary. Aligning variables on 32-bit boundaries produces code that runs somewhat faster on processors with 32-bit buses at the expense of more memory. Specifying this option aligns structures containing these datatypes differently than most published application binary interface specifications for the 68000 processor.

`-mbitfield`: Specifying this option tells GCC to use the bit-field instructions. The `-m68020` option implies this option. This is the default if you use a configuration designed for a 68020 processor.

`-mcfv4e`: Specifying this option tells GCC to generate code for processors in the ColdFire V4e family (547X, 548X), including hardware floating-point instructions. (This option is not available in the GCC 4.x compilers.)

`-mcpu32`: Specifying this option tells GCC to generate code for a CPU32 processor core. This is the default when the compiler is configured for CPU32-based systems. You should use this option when compiling code for microcontrollers with a CPU32 or CPU32+ core, including the 68330, 68331, 68332, 68333, 68334, 68336, 68340, 68341, 68349, and 68360 processors. The `-mcpu32` option implies the `-mnobitfield` option.

`-mfpa`: Specifying this option tells GCC to generate code that uses the Sun FPA instructions for floating point.

`-mid-shared-library`: Specifying this option causes GCC to generate code that supports the shared library ID mechanism. This mechanism supports both execute in place and the use of shared libraries without requiring virtual memory management. Using this option implies the `-fPIC` option.

`-mno-align-int`: Specifying this option tells GCC to align `int`, `long`, `long long`, `float`, `double`, and `long double` variables on a 16-bit boundary. This is the default.

`-mno-id-shared-library`: Specifying this option causes GCC to generate code that does not use the shared library ID mechanism. This is the default.

`-mno-sep-data`: Specifying this option causes GCC to assume that the data segment follows the text segment. This is the default.

`-mno-strict-align`: Specifying this option tells GCC to assume that unaligned memory references will be handled by the system.

`-mnobitfield`: Specifying this option tells GCC not to use the bit-field instructions. The `-m68000`, `-mcpu32`, and `-m5200` options imply the `-mnobitfield` option.

`-mpcrel`: Specifying this option tells GCC to use the PC-relative addressing mode of the 68000 directly, instead of using a global offset table. This option implies the standard GCC `-fpic` option, which therefore allows, at most, a 16-bit offset for PC-relative addressing. The `-fpic` option is not presently supported with the `-mpcrel` option.

`-mrtd`: Specifying this option tells GCC to use a function-calling convention where functions that take a fixed number of arguments return with the rtd instruction, which pops their arguments during the return. The rtd instruction is supported by the 68010, 68020, 68030, 68040, 68060, and CPU32 processors, but not by the 68000 or 5200 processors. Using the rtd instruction saves one instruction in the caller since there is no need to pop the arguments there. This calling convention is incompatible with the one normally used on Unix and therefore cannot be used if you need to call libraries that have been compiled with generic Unix compilers. When using this option, you must provide function prototypes for all functions that take variable numbers of arguments (including `printf`). If you do not, incorrect code will be generated for calls to those functions. You must also be careful to never call functions with extra arguments, which would result in seriously incorrect code.

■ **Tip** To optimize heavily used functions, you can specify that an individual function is called with the calling sequence specified by the `-mrtd option` with the function attribute `stdcall`. You can also override the `-mrtd` option for specific functions by using the function attribute `cdecl`.

`-msep-data`: Specifying this option tells GCC to generate code that allows the data segment to be located in a different memory area than the text segment. This option implies `-fPIC`, and enables support for execute in place without requiring virtual memory management.

`-mshared-library-id=`*n*: Specifying this option enables you to indicate the ID number (*n*) of an ID-based shared library that is being compiled.

`-mshort`: Specifying this option tells GCC to consider type `int` to be 16 bits wide, like `short int`.

`-msoft-float`: Specifying this option tells GCC that floating-point support should not be assumed and to generate output that contains library calls for floating-point operations. These libraries are not provided as part of GCC (except for the embedded GCC build targets m68k-*-aout and m68k-*-coff), but are normally found on the target system and should be resolved by the C loader on the target machine if they are available. When using this option and cross-compiling, you must provide suitable libraries containing at least function stubs on the host system, or these function calls will be identified as unresolved.

`-mstrict-align`: Specifying this option tells GCC not to assume that unaligned memory references will be handled by the system and to perform the alignment itself.

M68HC1x Options

The M68HC1x is a 16-bit microprocessor that is typically used as a microcontroller in embedded applications.

GCC options available when compiling code for M68HC1x systems are the following:

`-m6811` | `-m68hc11`: Specifying either of these options tells GCC to generate code for a 68HC11 microcontroller. This is the default when the compiler is configured for 68HC11-based systems.

`-m6812` | `-m68hc12`: Specifying either of these options tells GCC to generate code for a 68HC12 microcontroller. This is the default when the compiler is configured for 68HC12-based systems.

`-m68S12` | `-m68hcs12`: Specifying this option tells GCC to generate code for a 68HC12 microcontroller.

`-mauto-incdec`: Specifying this option tells GCC to use 68HC12 pre- and post-autoincrement and autodecrement addressing modes.

`-mlong-calls`: Specifying this option tells GCC to treat all calls as being far away, which causes GCC to use the call instruction to call a function, and the rtc instruction to return from those calls.

`-minmax`: Specifying this option enables GCC to use the 68HC12 min and max instructions.

`-mno-long-calls`: Specifying this option tells GCC to treat all calls as being near.

`-mno-minmax`: Specifying this option prevents GCC from using the 68HC12 min and max instructions.

`-mshort`: Specifying this option tells GCC to consider the `int` datatype to be 16 bits wide, like `short int`.

`-msoft-reg-count=`*count*: Specifying this option enables you to tell GCC the number of pseudosoft registers (*count*) that are used for the code generation. The maximum number is 32. Depending on the program, using more pseudosoft registers may or may not result in improved performance. The default is `4` for 68HC11, and `2` for 68HC12.

M88K Options

The Motorola 88000 family of RISC processors were primarily designed for use in workstations such as the Data General AViiON.

■ **Note** Support for this processor family is not provided in GCC version 4.0. These options are therefore only of interest if you are using a version of GCC that is earlier than 4.x and that you are certain provides support for this processor family.

GCC options available when compiling code for M88K systems are the following:

`-m88000`: Specifying this option tells GCC to generate code that works well on both the M88100 and the M88110 processors.

`-m88100`: Specifying this option tells GCC to generate code that works best for the M88100 processor, but that also runs on the M88110 processor.

`-m88110`: Specifying this option tells GCC to generate code that works best for the M88110 processor and may not run on the M88100 processor.

`-mbig-pic`: This deprecated option has the same effect as the `-fPIC` option on M88000 systems.

`-mcheck-zero-division`: Specifying this option tells GCC to generate code that guarantees that integer division by zero will be detected. This is the default. Some M88100 processors do not correctly detect integer division by zero, though all M88110 processors do. Using this option as a default generates code that will execute correctly on systems using either type of processor. This option is ignored if the `-m88110` option is specified.

`-mhandle-large-shift`: Specifying this option tells GCC to generate code that detects bit shifts of more than 31 bits and emits code to handle them properly.

`-midentify-revision`: Specifying this option tells GCC to include an `ident` directive in the assembler output that identifies the name of the source file, identifies the name and version of GCC, provides a time stamp, and records the GCC options used.

`-mno-check-zero-division`: Specifying this option tells GCC not to generate code that guarantees that integer division by zero will be detected. The MC88100 processor does not always trap on integer division by zero, so GCC generates additional code to explicitly check for zero divisors and trap with exception 503 when this is detected. This option is useful to reduce code size and possibly increase performance when running on systems with an MC88110 processor, which correctly detects all instances of integer division by zero.

`-mno-ocs-debug-info`: Specifying this option tells GCC not to include additional debugging information (about registers used in each stack frame) as specified in the 88open Object Compatibility Standard. This is the default on M88K systems running operating systems other than DG/UX, SVr4, and Delta 88 SVr3.2.

`-mno-ocs-frame-position`: Specifying this option tells GCC to use the offset from the frame pointer register (register `30`) when emitting COFF debugging information for automatic variables and parameters stored on the stack. When this option is in effect, the frame pointer is not eliminated when debugging information is selected by the `-g` switch. This is the default on M88K systems running operating systems other than DG/UX, SVr4, Delta 88 SVr3.2, and BCS.

`-mno-optimize-arg-area`: Specifying this option tells GCC not to reorganize the stack frame to save space, which results in increased memory use by the application. This is the default. The generated code conforms to the 88open Object Compatibility Standard specification.

`-mno-serialize-volatile`: Specifying this option tells GCC not to generate code to guarantee the sequential consistency of volatile memory references. This option is useful because by default GCC generates code to guarantee serial consistency, even on the M88100 processor, where serial consistency is guaranteed. This is done to enable the same code to run on M88110 systems, where the order of memory references does not always match the order of the instructions requesting those references. For example, on an M881100 processor, a load instruction may execute before a preceding store instruction. If you intend to run your code only on the M88100 processor, using the `-mno-serialize-volatile` option will produce smaller, faster code.

`-mno-underscores`: Specifying this option tells GCC to emit symbol names in assembler output without adding an underscore character at the beginning of each name. This is used for integration with linkers that do not follow the standard name mangling conventions. The default is to use an underscore as a prefix on each name.

`-mocs-debug-info`: Specifying this option tells GCC to include additional debugging information (about registers used in each stack frame) as specified in the 88open Object Compatibility Standard. This extra information allows debugging of code that has had the frame pointer eliminated. This is the default for M88K systems running DG/UX, SVr4, and Delta 88 SVr3.2.

`-mocs-frame-position`: Specifying this option tells GCC to use the offset from the canonical frame address when emitting COFF debugging information for automatic variables and parameters stored on the stack. The canonical frame address is the stack pointer (register `31`) on entry to the function. This is the default on M88K systems running DG/UX, SVr4, Delta 88 SVr3.2, and BCS.

`-moptimize-arg-area`: Specifying this option tells GCC to save space by reorganizing the stack frame, reducing memory consumption. This option generates code that does not agree with the 88open Object Compatibility Standard specification.

`-mserialize-volatile`: Specifying this option tells GCC to generate code that guarantees the serial consistency of volatile memory references. This is the default.

`-mshort-data-`*`num`*: Specifying this option tells GCC to generate smaller data references by making them relative to `r0`, which allows loading a value using a single instruction (rather than the usual two). The value specified for *`num`* enables you to control which data references are affected by this option by identifying the maximum displacement of short references that will be handled in this fashion. For example, specifying the `-mshort-data-512` option limits affected data references to those involving displacements of less than 512 bytes. The maximum value for `num` is 64K.

`-msvr3`: Specifying this option tells GCC to turn off compiler extensions related to System V Release 4 (SVR4). This option is the default for all GCC M88K build configurations other than the m88k-motorola-sysv4 and m88k-dg-dgux m88k GCC build targets.

`-msvr4`: Specifying this option tells GCC to turn on compiler extensions related to SVR4. Using this option makes the C preprocessor recognize `#pragma weak` and causes GCC to issue additional declaration directives that are used in SVR4. This option is the default for the m88k-motorola-sysv4 and m88k-dg-dgux m88k GCC build targets.

`-mtrap-large-shift`: Specifying this option tells GCC to generate code that traps on bit shifts of more than 31 bits. This is the default.

`-muse-div-instruction`: Specifying this option tells GCC to use the div instruction for signed integer division on the M88100 processor. By default, the div instruction is not used. On the M88100 processor, the signed integer division instruction traps to the operating system on a negative operand. The operating system transparently completes the operation, but at a large cost in execution time. On the M88110 processor, the div instruction (also known as the divs instruction) processes negative operands without trapping to the operating system. Using this option causes GCC to generate code that will run correctly on either type of processor. This option is ignored if the `-m88110` option is specified.

■ **Note** The result of dividing `int_min()`by -1 is undefined. In particular, the behavior of such a division with and without using the `-muse-div-instruction` option may differ.

`-mversion-03.00`: This option is obsolete and is ignored.

`-mwarn-passed-structs`: Specifying this option tells GCC to display a warning when a function passes a `struct` as an argument or a result. Structure-passing conventions have changed during the evolution of the C language and are often the source of portability problems. By default, GCC does not issue a warning.

MCore Options

Motorola's MCore processor family is a family of general-purpose 32-bit microcontrollers. Also known as M-Core or M*Core processors, the MCore family of processors is often used in embedded devices such as those employed for industrial control and measurement, health care and scientific equipment, and security systems.

GCC options available when compiling code for MCore-based systems are the following:

`-m210`: Specifying this option tells GCC to generate code for the 210 processor.

`-m340`: Specifying this option tells GCC to generate code for the 340 processor.

`-m4byte-functions`: Specifying this option tells GCC to force all functions to be aligned to a 4-byte boundary.

`-mbig-endian`: Specifying this option tells GCC to generate code for a big endian target.

`-mcallgraph-data`: Specifying this option tells GCC to emit callgraph information.

`-mdiv`: Specifying this option tells GCC to use the divide instruction. This is the default.

`-mhardlit`: Specifying this option tells GCC to inline constants in the code stream if it can be done in two instructions or less.

`-mlittle-endian`: Specifying this option tells GCC to generate code for a little endian target.

`-mno-4byte-functions`: Specifying this option tells GCC not to force all functions to be aligned to a 4-byte boundary.

`-mno-callgraph-data`: Specifying this option tells GCC not to emit callgraph information.

`-mno-div`: Specifying this option tells GCC not to use the divide instruction, replacing it with subtraction/remainder operations.

`-mno-hardlit`: Specifying this option tells GCC not to inline constants in the code stream.

`-mno-relax-immediate`: Specifying this option tells GCC not to allow arbitrarily sized immediates in bit operations.

`-mno-slow-bytes`: Specifying this option tells GCC not to prefer word access when reading byte quantities.

`-mno-wide-bitfields`: Specifying this option tells GCC not to treat bit fields as `int`-sized.

`-mrelax-immediate`: Specifying this option tells GCC to allow arbitrarily sized immediates in bit operations.

`-mslow-bytes`: Specifying this option tells GCC to prefer word access when reading byte quantities.

`-mwide-bitfields`: Specifying this option tells GCC to treat bit fields as `int`-sized.

MIPS Options

MIPS, an acronym for Microprocessor without Interlocked Pipeline Stages, is a microprocessor architecture developed by MIPS Computer Systems Inc. based on research and development done at Stanford University. The MIPS R2000 and R3000 processors were 32-bit processors, while later processors such as the R5000, R8000, R10000, R12000, and R16000 are all 64-bit processors. The R6000 processor was a third-party R3000 that quickly vanished, while the R7000 was targeted for embedded use and never saw wide deployment. Processors based on various MIPS cores are widely used in embedded systems.

GCC options available when compiling code for MIPS-based systems are the following:

`-EB`: Specifying this option tells GCC to compile code for the processor in big endian mode. The required libraries are assumed to exist.

`-EL`: Specifying this option tells GCC to generate code for the processor in little endian mode. The required libraries are assumed to exist.

`-G num`: Specifying this option tells GCC to put global and static objects less than or equal to `num` bytes into the small data or bss sections instead of the normal data or bss sections. Using this option enables the assembler to emit one-word memory reference instructions based on the global pointer (`GP` or `$28`), instead of using the normal two words. By default, `num` is `8` when the MIPS assembler is used, and `0` when the GNU assembler is used. The `-G num` switch is also passed to the assembler and linker.

■ **Note** All modules should be compiled with the same `-G num` value. Compiling with different values of `num` may or may not work. If it does not, the linker will display an error message and exit.

`-m4650`: Specifying this option is a convenient shortcut for the `-msingle-float`, `-mmad`, and `-march=r4650` options. (This option is not available in the GCC 4.x compilers.)

`-mabi=32`: Specifying this option tells GCC to generate code for the 32-bit ABI. The default instruction level is `-mips1` when using this option.

`-mabi=n32`: Specifying this option tells GCC to generate code for the new 32-bit ABI.

`-mabi=64`: Specifying this option tells GCC to generate code for the 64-bit application ABI. The default instruction level is `-mips4` when using this option.

`-mabi=o64`: Specifying this option tells GCC to generate code for the old 64-bit ABI.

`-mabi=eabi`: Specifying this option tells GCC to generate code for the EABI. The default instruction level is `-mips4` when using this option. By default, GCC generates 64-bit code for any 64-bit architecture, but you can specify the `-mgp32` option to force GCC to generate 32-bit code.

`-mabi=n32`: Specifying this option tells GCC to generate code for the new 32-bit ABI. The default instruction level is `-mips3` when using this option.

`-mabi=o64`: Specifying this option tells GCC to generate code for the old 64-bit ABI. The default instruction level is `-mips4` when using this option.

`-mabicalls`: Specifying this option tells GCC to generate code containing the pseudo-operations `.abicalls`, `.cpload`, and `.cprestore` that some SVR4 ports use for position-independent code. All code generated using the `-mabicalls` option is position-independent; this behavior can be changed to support absolute addresses for locally bound symbols and make direct calls to locally defined functions by specifying the `-mno-shared` option.

`-march=`*arch*: Specifying this option tells GCC to generate code for *arch*, which is either a MIPS ISA (Instruction Set Architecture) or a specific type of processor. Valid ISA values for *arch* are `mips1`, `mips2`, `mips3`, `mips4`, `mips32`, `mips32r2`, and `mips64`. Valid processor names are `4kc`, `4km`, `4kp`, `5kc`, `5kf`, `20kc`, `24k`, `24kc`, `24kf`, `24kx`, `m4k`, `orion`, `r2000`, `r3000`, `r3900`, `r4000`, `vr4100`, `vr4111`, `vr4120`, `vr4130`, `vr4300`, `r4400`, `r4600`, `r4650`, `vr5000`, `vr5400`, `vr5500`, `r6000`, `rm7000`, `r8000`, `rm9000`, `sr71000`, and `sb1`. The `r2000`, `r3000`, `r4000`, `r5000`, and `r6000` values can be abbreviated as `r2k` (or `r2K`), `r3k`, and so on, and the leading `vr` prefix can simply be abbreviated as `r`. When specifying an ISA, you can also abbreviate the entire `-march=`*arch* option as `-`*arch*. For example, `-mips32` is a valid synonym for `-march=mips32`.

■ **Tip** GCC uses the `arch` value specified using `-march=`*arch* to define two macros: `_MIPS_ARCH`, which provides the name of the target architecture as a string, and `_MIPS_ARCH_`*FOO*, where *FOO* is the name of the uppercase value of `_MIPS_ARCH`. The full numeric value for a specified processor is always used in these macros. For example, if you specify the argument `-march=r4k`, `_MIPS_ARCH` will be set to `r4000`, and the macro `_MIPS_ARCH_R4000` will be defined.

`-mbranch-likely`: Specifying this option tells GCC to use branch likely instructions on all architectures and processors where they are supported. This is the default for architectures other than MIPS32 and MIPS64, where the branch likely instructions are officially deprecated.

`-mcheck-zero-division`: Specifying this option tells GCC to trap on integer division by zero. This is the default.

`-mdivide-breaks`: Specifying this option tells GCC to check for integer division by zero by generating a break instruction. Checking for integer division by zero can be disabled entirely by specifying the `-mno-check-zero-division` option.

`-mdivide-traps`: Specifying this option tells GCC to check for integer division by zero by generating a conditional trap. This is only supported on MIPS 2 or later processors. This option is the default unless GCC was configured using the `--with-divide=breaks` option. Checking for integer division by zero can be disabled entirely by specifying the `-mno-check-zero-division` option.

`-mdouble-float`: Specifying this option tells GCC to assume that the floating-point coprocessor supports double-precision operations. This is the default.

`-mdsp`: Specifying this option enables GCC to generate code that uses the MIPS DSP ASE (digital signal processor application-specific extension).

`-membedded-data`: Specifying this option tells GCC to allocate variables in the read-only data section first, if possible, then in the small data section, if possible, or finally in the data section. This results in slightly slower code than the default, but reduces the amount of RAM required when executing, and thus may be preferred for some embedded systems.

`-membedded-pic`: Specifying this option tells GCC to generate PIC code that is suitable for embedded systems. All calls are made using PC-relative addresses, and all data is addressed using the `$gp` register. No more than 65,536 (64K) bytes of global data may be used. Using this option requires GNU as and GNU ld, which do most of the work. This option currently only works on targets that use the ECOFF binary output format. It does not work with the ELF binary output format.

`-mentry`: Specifying this option tells GCC to use the entry and exit pseudo-ops. This option can only be used with the `-mips16` option.

`-mexplicit-relocs`: Specifying this option tells GCC to use assembler relocation operators when resolving symbolic addresses. This option is the default if GCC was configured to use an assembler that supports relocation operators, such as the GNU assembler.

`-mfix-r4000`: Specifying this option tells GCC to use workarounds for various problems with the R4000 processor.

`-mfix-r4400`: Specifying this option tells GCC to use workarounds for various problems with the R4400 processor.

`-mfix-sb1`: Specifying this option tells GCC to use workarounds for various problems with the SB-1 CPU core.

`-mfix-vr4120`: Specifying this option tells GCC to use workarounds for various problems with the VR4120 processor.

`-mfix-vr4130`: Specifying this option tells GCC to use workarounds for various problems with the VR4130 processor.

`-mflush-func=func`: Specifying this option tells GCC the function to call in order to flush the I and D caches. The function must take the same arguments as the common `_flush_func()` function, which are the address of the memory range for which the cache is being flushed, the size of the memory range, and the number 3 (to flush both caches). The default is usually `_flush_func` or `__cpu_flush`, and is defined by the macro `TARGET_SWITCHES` in the machine description file.

`-mfp-exceptions`: Specifying this option enables GCC to use floating-point exceptions, which affects the scheduling of floating-point instructions on some processors. This option is the default.

`-mfp32`: Specifying this option tells GCC to assume that 32 32-bit floating-point registers are available. This is the default.

`-mfp64`: Specifying this option tells GCC to assume that 32 64-bit floating-point registers are available. This is the default when the `-mips3` option is used.

`-mfused-madd`: Specifying this option tells GCC to generate code that uses the floating-point multiply and accumulate instructions when they are available. These instructions are generated by default if they are available.

`-mgas`: Specifying this option tells GCC to generate code for the GNU assembler. This is the default on the OSF/1 reference platform, which uses the OSF/rose object format. More significantly, this is the default if the GCC configuration option `--with-gnu-as` is used, which is the GCC default. (This option is not available in the GCC 4.x compilers.)

`-mgp32`: Specifying this option tells GCC to assume that 32 32-bit general-purpose registers are available. This is the default.

`-mgp64`: Specifying this option tells GCC to assume that 32 64-bit general-purpose registers are available. This is the default when the `-mips3` option is used.

`-mgpopt`: Specifying this option tells GCC to write all of the data declarations before the instructions in the text section. This allows the MIPS assembler to generate one-word memory references instead of using two words for short global or static data items. This is the default if optimization is selected. (This option is not available in the GCC 4.x compilers.)

`-mhalf-pic`: Specifying this option tells GCC to put pointers to `extern` references into the data section, rather than putting them in the text section. (This option is not available in the GCC 4.x compilers.)

`-mhard-float`: Specifying this option tells GCC to generate code that contains floating-point instructions. This is the default.

`-mint64`: Specifying this option tells GCC to force `int` and `long` types to be 64 bits wide. See the discussion of the `-mlong32` option for an explanation of the default and the width of pointers. (This option is not available in the GCC 4.x compilers.)

`-mips1`: Specifying this option tells GCC to generate instructions that are conformant to level 1 of the MIPS ISA. This is the default. `r3000` is the default `CPU-type` at this ISA level. The default ABI is 32 (`-mabi=32`) when using this option.

`-mips16`: Specifying this option tells GCC to enable the use of 16-bit instructions.

`-mips2`: Specifying this option tells GCC to generate instructions that are conformant to level 2 of the MIPS ISA (branch likely, square root instructions, etc.). `r6000` is the default `CPU-type` at this ISA level. The default ABI is 32 (`-mabi=32`) when using this option.

`-mips3`: Specifying this option tells GCC to generate instructions that are conformant to level 3 of the MIPS ISA (64-bit instructions). `r4000` is the default `CPU-type` at this ISA level. The default ABI is 64 (`-mabi=64`) when using this option.

`-mips3d`: Specifying this option enables GCC to generate code that uses the MIPS 3D ASE (three-dimensional application-specific extensions).

`-mips4`: Specifying this option tells GCC to generate instructions that are conformant to level 4 of the MIPS ISA (conditional move, prefetch, enhanced FPU instructions). `r8000` is the default `CPU-type` at this ISA level. The default ABI is 64 (`-mabi=64`) when using this option.

`-mlong32`: Specifying this option tells GCC to force `long`, `int`, and `pointer` types to be 32 bits wide.

Note If none of the options `-mlong32`, `-mlong64`, or `-mint64` are set, the size of `int`s, `long`s, and `pointer`s depends on the ABI and ISA chosen. For `-mabi=32` and `-mabi=n32`, `int`s and `long`s are 32 bits wide. For `-mabi=64`, `int`s are 32 bits wide, and `long`s are 64 bits wide. For `-mabi=eabi` and either `-mips1` or `-mips2`, `int`s and `long`s are 32 bits wide. For `-mabi=eabi` and higher ISAs, `int`s are 32 bits, and `long`s are 64 bits wide. The width of pointer types is the smaller of the width of `long`s or the width of general-purpose registers (which in turn depends on the ISA).

`-mlong64`: Specifying this option tells GCC to force `long` types to be 64 bits wide. See the discussion of the `-mlong32` option for an explanation of the default and the width of pointers.

`-mlong-calls`: Specifying this option tells GCC to make all function calls using the JALR instruction, which requires loading a function's address into a register before making the call. You must use this option if you call outside of the current 512-megabyte segment to functions that are not called through pointers.

`-mmad`: Specifying this option tells GCC to permit the use of the mad, madu, and mul instructions, as on the r4650 chip.

`-mmemcpy`: Specifying this option tells GCC to make all block moves call the appropriate string function (`memcpy()` or `bcopy()`) instead of possibly generating inline code.

`-mno-fp-exceptions`: Specifying this option prevents GCC from using floating-point exceptions.

`-mno-fused-madd`: Specifying this option prevents GCC from generating code that uses the floating-point multiply and accumulate instructions, even when they are available.

`-mno-mips3d`: Specifying this option prevents GCC from generating code that uses the MIPS 3D ASE. This option is the default.

`-mno-abicalls`: Specifying this option tells GCC not to generate code containing the pseudo-operations `.abicalls`, `.cpload`, and `.cprestore` that some System V.4 ports use for position-independent code.

`-mno-branch-likely`: Specifying this option prevents GCC from using the branch likely instruction.

`-mno-check-zero-division`: Specifying this option tells GCC not to check for attempts to perform integer division by zero.

`-mno-dsp`: Specifying this option prevents GCC from generating code that uses the MIPS DSP ASE. This option is the default.

`-mno-explicit-relocs`: Specifying this option tells GCC to use macros rather than assembler relocation operators when resolving symbolic addresses.

`-mno-flush-func`: Specifying this option tells GCC not to call any function to flush the I and D caches.

`-mno-fused-madd`: Specifying this option tells GCC not to generate code that uses the floating-point multiply and accumulate instructions, even if they are available. These instructions may be undesirable if the extra precision causes problems or on certain chips in the modes where denormals are rounded to zero and where denormals generated by multiply and accumulate instructions cause exceptions anyway.

`-mno-gpopt`: Specifying this option tells GCC not to write all of the data declarations before the instructions in the text section, preventing some optimizations but resulting in more readable assembly code.

`-mno-long-calls`: Specifying this option tells GCC not to use the JALR instruction when making function calls.

`-mno-mad`: Specifying this option tells GCC not to permit the use of the mad, madu, and mul instructions.

`-mno-memcpy`: Specifying this option tells GCC to possibly generate inline code for all block moves rather than calling the appropriate string function (`memcpy()` or `bcopy()`).

`-mno-mips-tfile`: Specifying this option tells GCC not to post-process the object file with the `mips-tfile` program, which adds debugging support after the MIPS assembler has generated it. If the `mips-tfile` program is not run, no local variables will be available to the debugger. In addition, `stage2` and `stage3` objects will have the temporary filenames passed to the assembler embedded in the object file, which means the objects will not compare the same. The `-mno-mips-tfile` switch should only be used when there are bugs in the `mips-tfile` program that prevents compilation. This option is the default in modern versions of GCC, which all use the GNU assembler.

`-mno-mips16`: Specifying this option tells GCC not to use 16-bit instructions.

`-mno-embedded-data`: Specifying this option tells GCC to allocate variables in the data section, as usual. This may result in faster code but increases the amount of RAM required to run an application.

`-mno-embedded-pic`: Specifying this option tells GCC not to make all function calls using PC-relative addresses, and not to address all data using the `$gp` register. This is the default on systems that use the ELF binary output format, or use an assembler or linker/loader other than the GNU tools. (This option is not available in the GCC 4.x compilers.)

`-mno-half-pic`: Specifying this option tells GCC to put pointers to `extern` references in the text section. This option is the default. (This option is not available in the GCC 4.x compilers.)

`-mno-rnames`: Specifying this option tells GCC to generate code that uses the hardware names for the registers (i.e., `$4`). This is the default. (This option is not available in the GCC 4.x compilers.)

`-mno-shared`: Specifying this option prevents GCC from generating position-independent code. Specifying this option also supports absolute addresses for locally bound symbols and direct calls to locally defined functions. (This option is not available in the GCC 4.x compilers.)

`-mno-split-addresses`: Specifying this option tells GCC not to generate code that loads the high and low parts of address constants separately, which prevents some optimizations but may be necessary if using a non-GNU assembler or linker/loader.

`-mno-stats`: Specifying this option tells GCC not to emit statistical information when processing noninline functions. This is the default. (This option is not available in the GCC 4.x compilers.)

`-mno-sym32`: Specifying this option tells GCC not to assume that all symbols have 32-bit values.

`-mno-uninit-const-in-rodata`: Specifying this option tells GCC to store uninitialized `const` variables in the data section, as usual. This is the default.

`-mno-vr4130-align`: Specifying this option prevents GCC from aligning pairs of instructions that might be able to execute in parallel. This produces smaller code, but doesn't take advantage of the VR4130's two-way superscalar pipeline.

`-mno-xgot`: Specifying this option tells GCC to impose a 64K limitation on the size of the GOT, which therefore only requires a single instruction to fetch the value of a global symbol. This option is the default.

`-mshared`: Specifying this option causes GCC to generate position-independent code that can be linked into shared libraries. All code generated using the `-mabicalls` option is position-independent—this behavior can be changed to support absolute addresses for locally bound symbols by specifying the `-mno-shared` option.

`-msingle-float`: Specifying this option tells GCC to assume that the floating-point coprocessor only supports single precision operations, as on the r4650 chip.

`-msoft-float`: Specifying this option tells GCC that floating-point support should not be assumed, and to generate output that contains library calls for floating-point operations. These libraries are not provided as part of GCC, but are normally found on the target system, and should be resolved by the C loader on the target machine if they are available. When using this option and cross-compiling, you must provide suitable libraries containing at least function stubs on the host system or these function calls will be identified as unresolved.

`-msplit-addresses`: Specifying this option tells GCC to generate code that loads the high and low parts of address constants separately. This allows GCC to optimize away redundant loads of the high order bits of addresses. This optimization requires GNU as and GNU ld. This optimization is enabled by default for some embedded targets where GNU as and GNU ld are standard.

`-mstats`: Specifying this option tells GCC to emit one line to the standard error output for each noninline function processed. This line provides statistics about the program (number of registers saved, stack size, and so on).

`-msym32`: Specifying this option tells GCC to assume that all symbols have 32-bit values, regardless of the ABI that you are using. This option can provide faster access to symbolic addresses when used in conjunction with the `-mabi=64` and `-no-abicalls` options.

`-mtune=`*`arch`*: Specifying this option tells GCC to use the defaults for the specified machine type *`arch`* when scheduling instructions. See the `-march=`*`arch`* option for a list of possible values for *`arch`*. Supplying different values for *`arch`* to the `-march=`*`arch`* and `-mtune=`*`arch`* options enables GCC to generate code that runs on an entire family of processors, but is tuned for optimal performance on a specific CPU. If this option is not specified, the code produced by GCC is tuned for the *`arch`* specified using any `-march=`*`arch`* option.

■ **Tip** GCC uses the `arch` value specified using `-mtune=`*`arch`* to define two macros: `_MIPS_TUNE`, which provides the name of the target architecture as a string, and `_MIPS_TUNE_`*`FOO`*, where *`FOO`* is the name of the uppercase value of `_MIPS_TUNE`. The full numeric value for a specified processor is always used in these macros. For example, if you specify the argument `-march=r4k`, `_MIPS_TUNE` will be set to `r4000`, and the macro `_MIPS_TUNE_R4000` will be defined.

`-muninit-const-in-rodata`: Specifying this option tells GCC to store uninitialized `const` variables in the read-only data section. This option must be used with the `-membedded-data` option.

`-mvr4130-align`: Specifying this option tells GCC to align pairs of instructions that it believes can execute in parallel in order to take advantage of the VR4130's two-way superscalar pipeline. This option is used when optimizing code for the VR4130, produces faster code at the expense of code size, and is active by default at optimization levels `-O3` or greater.

`-mxgot`: Specifying this option tells GCC to remove limitations on the size of the GOT, which therefore requires multiple instructions to fetch the value of a global symbol. This is necessary when the size of the GOT table exceeds 64K.

`-nocpp`: Specifying this option instructs the MIPS assembler not to run its preprocessor over user assembler files (files with an `.s` suffix) when assembling them.

`-no-crt0`: Specifying this option tells GCC not to include the default crt0 C runtime initialization library.

MMIX Options

MIX was a virtual computer system designed for use as an example in Donald Knuth's legendary *The Art of Computer Programming* books. MMIX is a 64-bit RISC-oriented follow-up to MIX.

GCC options available when compiling code for MMIX-based systems are the following:

`-mabi=gnu`: Specifying this option tells GCC to generate code that is conformant to the GNU ABI and therefore passes function parameters and return values in global registers `$231` and up.

`-mabi=mmixware`: Specifying this option tells GCC to generate code that passes function parameters and return values that (in the called function) are seen as registers `$0` and up.

`-mbase-addresses`: Specifying this option tells GCC to generate code that uses `_base addresses_`. Using a base address automatically generates a request (handled by the assembler and the linker) for a constant to be set up in a global register. The register is used for one or more base address requests within the range 0 to 255 from the value held in the register. This generally leads to short and fast code, but the number of different data items that can be addressed is limited. This means that a program that uses a lot of static data may require `-mno-base-addresses`.

`-mbranch-predict`: Specifying this option tells GCC to use the probable-branch instructions when static branch prediction indicates a probable branch.

`-melf`: Specifying this option tells GCC to generate code in ELF format, rather than in the default mmo format used by the MMIX simulator.

`-mepsilon`: Specifying this option tells GCC to generate floating-point comparison instructions that compare with respect to the `rE` epsilon register.

`-mknuthdiv`: Specifying this option tells GCC to make the result of a division yielding a remainder have the same sign as the divisor.

`-mlibfuncs`: Specifying this option tells GCC that all intrinsic library functions are being compiled, passing all values in registers, no matter the size.

`-mno-base-addresses`: Specifying this option tells GCC not to generate code that uses `_base addresses_`. Using a base address generally leads to short and fast code but limits the number of different data items that can be addressed. Programs that use significant amounts of static data may require `-mno-base-addresses`.

`-mno-branch-predict`: Specifying this option tells GCC not to use the probable-branch instructions.

`-mno-epsilon`: Specifying this option tells GCC not to generate floating-point comparison instructions that use the `rE` epsilon register.

`-mno-knuthdiv`: Specifying this option tells GCC to make the result of a division yielding a remainder have the same sign as the dividend. This is the default.

`-mno-libfuncs`: Specifying this option tells GCC that all intrinsic library functions are not being compiled and that values should therefore not be passed in registers.

`-mno-single-exit`: Specifying this option enables GCC to support multiple exit points in a function.

`-mno-toplevel-symbols`: Specifying this option tells GCC not to insert a colon (:) at the beginning of all global symbols. This disallows the use of the `PREFIX` assembly directive with the resulting assembly code.

`-mno-zero-extend`: Specifying this option tells GCC to use sign-extending load instructions when reading data from memory in sizes shorter than 64 bits.

`-msingle-exit`: Specifying this option tells GCC to force generated code to have a single exit point in each function.

`-mtoplevel-symbols`: Specifying this option tells GCC to insert a colon (:) at the beginning of all global symbols, so that the assembly code can be used with the `PREFIX` assembly directive.

`-mzero-extend`: Specifying this option tells GCC to use zero-extending load instructions when reading data from memory in sizes shorter than 64 bits.

MN10200 Options

The MN10200 series of 16-bit single-chip microcontrollers from Panasonic are low-power processors with a 16MB address space and fast instruction execution time complemented by a three-stage pipeline.

■ **Note** Support for this processor family is not provided in GCC version 4.0. This option is therefore only of interest if you are using a version of GCC that is earlier than 4.x and which you are certain provides support for this processor family.

The GCC option available when compiling code for MN10200-based systems is the following:

`-mrelax`: Specifying this option tells GCC to tell the linker that it should perform a relaxation optimization pass to shorten branches, calls, and absolute memory addresses. This option is only useful when specified on the command line for the final link step. Using this option makes symbolic debugging impossible.

MN10300 Options

The MN10300 series of 32-bit single-chip microcontrollers are the descendents of the MN10200 processors, and add built-in support for multimedia applications to the core capabilities of the MN10200.

GCC options available when compiling code for MN10300-based systems are the following:

`-mam33`: Specifying this option tells GCC to generate code that uses features specific to the AM33 processor.

`-mmult-bug`: Specifying this option tells GCC to generate code that avoids bugs in the multiply instructions for the MN10300 processors. This is the default.

`-mno-am33`: Specifying this option tells GCC not to generate code that uses features specific to the AM33 processor. This is the default.

`-mno-crt0`: Specifying this option tells GCC not to link in the C runtime initialization object file.

`-mno-mult-bug`: Specifying this option tells GCC not to generate code that avoids bugs in the multiply instructions for the MN10300 processors. This may result in smaller, faster code if your application does not trigger these bugs.

`-mno-return-pointer-on-d0`: Specifying this option tells GCC to only return a pointer in `a0` when generating code for functions that return pointers.

`-mrelax`: Specifying this option tells GCC to tell the linker that it should perform a relaxation optimization pass to shorten branches, calls, and absolute memory addresses. This option is only useful when specified on the command line for the final link step. Using this option makes symbolic debugging impossible.

`-mreturn-pointer-on-d0`: Specifying this option tells GCC to return pointers in both `a0` and `d0` when generating code for functions that return pointers. This option is the default.

MT Options

The Morpho Technologies MS1 and MS2 processors are reconfigurable digital signal processors that are popular in wireless and handheld devices.

GCC options available when compiling code for MT systems are the following:

`-march=`*`cpu-type`*: Specifying this option tells GCC to generate code that will run on *`cpu-type`*, which is a specific type of MT processor. Possible values for *`cpu-type`* are `ms1-64-001`, `ms1-16-002`, `ms1-16-003`, and `ms2`. If no `-march=`*`cpu-type`* option is specified, the default value is `-march=ms1-16-002`.

`-mbacc`: Specifying this option tells GCC to use byte loads and stores when generating code.

`-mno-bacc`: Specifying this option prevents GCC from using byte loads and stores when generating code.

`-mno-crt0`: Specifying this option prevents GCC from linking the object code that it generates with the C runtime initialization object file crti.o. Other runtime initialization and termination files, such as startup.o and exit.o, will still be used.

`-msim`: Specifying this option tells GCC to use a special runtime with the MT simulator.

NS32K Options

The National Semiconductor NS32000 (a.k.a. NS32K) family of processors was used in a variety of older computer systems and has also been used in embedded and signal processing applications.

■ **Note** Support for this processor family is not provided in GCC version 4.0. These options are therefore only of interest if you are using a version of GCC that is earlier than 4.x and that you are certain provides support for this processor family.

GCC options available when compiling code for NS32000-based systems are the following:

`-m32032`: Specifying this option tells GCC to generate code for a 32032 processor. This is the default when the compiler is configured for 32032- and 32016-based systems.

`-m32081`: Specifying this option tells GCC to generate code containing 32081 instructions for floating point. This is the default for all systems.

`-m32332`: Specifying this option tells GCC to generate code for a 32332 processor. This is the default when the compiler is configured for 32332-based systems.

`-m32381`: Specifying this option tells GCC to generate code containing 32381 instructions for floating point. Specifying this option also implies the `-m32081` option. The 32381 processor is only compatible with the 32332 and 32532 CPUs. This is the default for GCC's `pc532-netbsd` build configuration.

`-m32532`: Specifying this option tells GCC to generate code for a 32532 processor. This is the default when the compiler is configured for 32532-based systems.

`-mbitfield`: Specifying this option tells GCC to use bit-field instructions. This is the default for all platforms except the pc532.

`-mhimem`: Specifying this option tells GCC to generate code that can be loaded above 512MB. Many NS32000 series addressing modes use displacements of up to 512MB. If an address is above 512MB, then displacements from zero cannot be used. This option is often useful for operating systems or ROM code.

`-mmulti-add`: Specifying this option tells GCC to attempt to generate the multiply-add floating-point instructions polyF and dotF. This option is only available if the `-m32381` option is also specified. Using these instructions requires changes to register allocation that generally have a negative impact on performance. This option should only be enabled when compiling code that makes heavy use of multiply-add instructions.

`-mnobitfield`: Specifying this option tells GCC not to use the bit-field instructions. On some machines, such as the pc532, it is faster to use shifting and masking operations. This is the default for the pc532.

`-mnohimem`: Specifying this option tells GCC to assume that code will be loaded in the first 512MB of virtual address space. This is the default for all platforms.

`-mnomulti-add`: Specifying this option tells GCC not to generate code containing the multiply-add floating-point instructions polyF and dotF. This is the default on all platforms.

`-mnoregparam`: Specifying this option tells GCC not to pass any arguments in registers. This is the default for all targets.

`-mnosb`: Specifying this option tells GCC not to use the `sb` register as an index register. This is usually because it is not present or is not guaranteed to have been initialized. This option is the default for all targets except the pc532-netbsd. This option is also implied whenever the `-mhimem` or `-fpic` options are specified.

`-mregparam`: Specifying this option tells GCC to use a different function-calling convention where the first two arguments are passed in registers. This calling convention is incompatible with the one normally used on Unix, and therefore should not be used if your application calls libraries that have been compiled with the Unix compiler.

`-mrtd`: Specifying this option tells GCC to use a function-calling convention where functions that take a fixed number of arguments pop their arguments during the return. This calling convention is incompatible with the one normally used on Unix, and therefore cannot be used if you need to call libraries that have been compiled with generic Unix compilers. When using this option, you must provide function prototypes for all functions that take variable numbers of arguments (including `printf`); if you do not, incorrect code will be generated for calls to those functions. You must also be careful to never call functions with extra arguments, which would result in seriously incorrect code. This option takes its name from the 680x0 rtd instruction.

`-msoft-float`: Specifying this option tells GCC that floating-point support should not be assumed and to generate output that contains library calls for floating-point operations. These libraries are not provided as part of GCC, but are normally found on the target system and should be resolved by the C loader on the target machine if they are available. When using this option and cross-compiling, you must provide suitable libraries containing at least function stubs on the host system or these function calls will be identified as unresolved.

`-msb`: Specifying this option tells GCC that the `sb` can be used as an index register that is always loaded with zero. This is the default for the `pc532-netbsd` target.

PDP-11 Options

The options in this section are specific to using GCC to compile applications for the former Digital Equipment Corporation's PDP-11 minicomputers. Few of these systems are still in use, aside from some that are still used in older process control applications. However, these options can still be useful if you happen to have PDP-11 in the basement "just in case." I know I do.

GCC options available when compiling code for PDP-11 systems are the following:

`-m10`: Specifying this option tells GCC to generate code for a PDP-11/10.

`-m40`: Specifying this option tells GCC to generate code for a PDP-11/40.

`-m45`: Specifying this option tells GCC to generate code for a PDP-11/45. This is the default.

`-mabshi`: Specifying this option tells GCC to use the abshi2 pattern. This is the default.

`-mac0`: Specifying this option tells GCC to return floating-point results in `ac0` (`fr0` in Unix assembler syntax).

`-mbcopy`: Specifying this option tells GCC not to use inline movstrhi patterns for copying memory.

`-mbcopy-builtin`: Specifying this option tells GCC to use inline movstrhi patterns for copying memory. This is the default.

`-mbranch-cheap`: Specifying this option tells GCC not to assume that branches are expensive. This is the default.

`-mbranch-expensive`: Specifying this option tells GCC to assume that branches are expensive. This option is designed for experimenting with code generation and is not intended for production use.

`-mdec-asm`: Specifying this option tells GCC to use DEC (Compaq? HP?) assembler syntax. This is the default when GCC is configured for any PDP-11 build target other than `pdp11-*-bsd`.

`-mfloat32 | -mno-float64`: Specifying either of these options tells GCC to use 32-bit floats.

`-mfloat64 | -mno-float32`: Specifying either of these options tells GCC to use 64-bit floats. This is the default.

`-mfpu`: Specifying this option tells GCC to use hardware FPP floating point. This is the default. (FIS floating point on the PDP-11/40 is not supported.)

`-mint16 | -mno-int32`: Specifying either of these options tells GCC to use 16-bit `int`s. This is the default.

`-mint32 | -mno-int16`: Specifying either of these options tells GCC to use 32-bit `int`s.

`-mno-abshi`: Specifying this option tells GCC not to use the abshi2 pattern.

`-mno-ac0`: Specifying this option tells GCC to return floating-point results in memory. This is the default.

`-mno-split`: Specifying this option tells GCC to generate code for a system without split instruction and data spaces. This is the default.

`-msoft-float`: Specifying this option tells GCC that floating-point support should not be assumed, and to generate output that contains library calls for floating-point operations. These libraries are not provided as part of GCC, but are normally found on the target system, and should be resolved by the C loader on the target machine if they are available. When using this option and cross-compiling, you must provide suitable libraries containing at least function stubs on the host system or these function calls will be identified as unresolved.

`-msplit`: Specifying this option tells GCC to generate code for a system with split Instruction and Data spaces.

`-munix-asm`: Specifying this option tells GCC to use Unix assembler syntax. This is the default when GCC is configured for the `pdp11-*-bsd` build target.

PowerPC (PPC) Options

The PowerPC is a RISC microprocessor architecture that was originally created by the 1991 Apple-IBM-Motorola alliance, known as AIM. The PowerPC was the CPU portion of the overall AIM platform, and is the only surviving aspect of the platform. Performance Optimization With Enhanced RISC (POWER) had been introduced (and was doing well) as a multichip processor on the IBM RS/6000 workstation, but IBM was interested in a single-chip version as well as entering other markets. At the same time, Apple was looking for a new processor to replace the aging MC680x0 processors in its Macintosh computers.

GCC provides options that enable you to specify which instructions are available on the processor you are using. GCC supports two related instruction set architectures for the PowerPC and RS/6000. The POWER instruction set consists of those instructions that are supported by the RIOS chip set used in the original RS/6000 systems. The PowerPC instruction set is the architecture of the Motorola MPC5xx, MPC6xx, MPC8xx microprocessors, and the IBM 4xx microprocessors. Neither architecture is a subset of the other, but both support a large common subset of instructions. An `mq` register is included in processors supporting the POWER architecture.

■ **Tip** In general, it is easier to use the `-mcpu=CPU-type` option, which implies the appropriate instruction set, rather than trying to remember the appropriate `-mpower*` option.

IBM's RS64 processor family is a modified PowerPC architecture. These processors are used in the AS/400 computer family and in some RS/6000 systems. The latest generation of PowerPC processors (the G5) was used in Apple Macintosh computer systems prior to Apple's switch to Intel processors. PowerPC chips based on older cores are tremendously popular in embedded hardware.

GCC options available when compiling code for PowerPC systems are the following:

`-G num`: Specifying this option on embedded PowerPC systems tells GCC to put global and static objects less than or equal to `num` bytes into the small data or bss sections instead of the normal data or bss sections. By default, `num` is `8`. The `-G num` switch is also passed to the assembler and linker.

■ **Note** All modules should be compiled with the same `-G num` value. Compiling with different values of `num` may or may not work. If it does not, the linker will display an error message and exit.

`-m32`: Specifying this option tells GCC to generate code for a 32-bit environment. The 32-bit environment sets `int`, `long`, and `pointer` to 32 bits and generates code that will run on any PowerPC processor.

`-m64`: Specifying this option tells GCC to generate code for a 64-bit environment. The 64-bit environment sets `int` to 32 bits and `long` and `pointer` to 64 bits and generates code that will only run on PowerPC-64 processors.

`-mabi=`*`abi-type`*: Specifying this option tells GCC to extend the current ABI with a specific set of extensions or to remove a specified set of extensions. Possible values for *`abi-type`* are `altivec` (adds AltiVec vector-processing ABI extensions), `ibmlongdouble` (adds IBM extended precision `long double` extensions, a PowerPC 32-bit SYSV ABI option), `ieeelongdouble` (adds IEEE extended precision `long double` extensions, a PowerPC 32-bit Linux ABI option), `no-altivec` (disables AltiVec ABI extensions for the current ABI), `no-spe` (disables IBM's Synergistic Processing Elements [SPE] SIMD instructions for the current ABI), and `spe` (adds SPE SIMD instructions). Using this option does not change the current ABI except to add or remove the specified extensions.

`-mads`: Specifying this option on embedded PowerPC systems tells GCC to assume that the startup module is called crt0.o and the standard C libraries are libads.a and libc.a.

`-maix-struct-return`: Specifying this option tells GCC to return all structures in memory (as specified by the AIX ABI).

`-maix32`: Specifying this option tells GCC to use the 32-bit ABI, disabling the 64-bit AIX ABI and calling conventions. This is the default.

`-maix64`: Specifying this option tells GCC to enable the 64-bit AIX ABI and calling convention: 64-bit pointers, 64-bit `long` type, and the infrastructure needed to support them. Specifying `-maix64` implies the `-mpowerpc64` and `-mpowerpc` options.

`-malign-natural`: Specifying this option tells GCC to override the alignment of larger types on natural size-based boundaries, as defined by the ABI. This option can be used on AIX, 32-bit Darwin, and 64-bit PowerPC Linux systems. This is the default on 64-bit Darwin systems.

`-malign-power`: Specifying this option tells GCC to follow the alignment rules for larger types as specified in the ABI. This option is not supported on 64-bit Darwin systems.

`-maltivec`: Specifying this option tells GCC to enable the use of built-in functions that provide access to the AltiVec instruction set. You may also need to set `-mabi=altivec` to adjust the current ABI with AltiVec ABI enhancements.

`-mbig|-mbig-endian`: Specifying either of these options on SVR4 or embedded PowerPC systems tells GCC to compile code for the processor in big endian mode.

`-mbit-align`: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to force structures and unions that contain bit fields to be aligned to the base type of the bit field. This option is the default. For example, by default a structure containing nothing but eight unsigned bit fields of length 1 would be aligned to a 4-byte boundary and have a size of 4 bytes.

`-mbss-plt`: Specifying this option enables GCC to use a bss PLT section that is filled in by the linker, and requires PLT and GOT sections that are both writable and executable. This is a PowerPC 32-bit SYSV ABI option.

`-mcall-aix`: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to compile code using calling conventions that are similar to those used on AIX. This is the default if you configure GCC using the `powerpc-*-eabiaix` GCC build target. (This option is not available in the GCC 4.x compilers.)

`-mcall-gnu`: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to compile code using calling conventions similar to those used by the HURD-based GNU system.

`-mcall-linux`: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to compile code using calling conventions similar to those used by the Linux-based GNU system.

`-mcall-netbsd`: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to compile code using calling conventions similar to those used by the NetBSD operating system.

`-mcall-solaris`: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to compile code using calling conventions that are similar to those used by the Solaris operating system.

`-mcall-sysv`: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to compile code using calling conventions that adhere to the March 1995 draft of the PowerPC processor supplement to the System V application binary interface. This is the default unless you configured GCC using the `powerpc-*-eabiaix` GCC build target.

`-mcall-sysv-eabi`: Specifying this option is the same as using both the `-mcall-sysv` and `-meabi` options.

`-mcall-sysv-noeabi`: Specifying this option is the same as using both the `-mcall-sysv` and `-mno-eabi` options.

`-mcpu=`*`cpu-type`*: Specifying this option tells GCC to set the architecture type, register usage, choice of mnemonics, and instruction scheduling parameters to values associated with the machine type *`CPU-type`*. Possible values for *`cpu-type`* are `401`, `403`, `505`, `405fp`, `440`, `440fp`, `505`, `601`, `602`, `603`, `603e`, `604`, `604e`, `620`, `630`, `740`, `7400`, `7450`, `750`, `801`, `821`, `823`, `860`, `970`, `common`, `power`, `power2`, `power3`, `power4`, `power5`, `power5+`, `powerpc`, `powerpc64`, `rios`, `rios1`, `rsc`, `rios2`, `rsc`, and `rs64a`. The `-mcpu=power`, `-mcpu=power2`, `-mcpu=powerpc`, and `-mcpu=powerpc64` options specify generic POWER, POWER2, pure 32-bit PowerPC, and 64-bit PowerPC architecture machine types, with an appropriate, generic processor model assumed for scheduling purposes. The other options specify a specific processor. Code generated under those options will run best on that processor and may not run at all on others.

■ **Tip** Specifying the `-mcpu=common` option selects a completely generic processor. Code generated under this option will run on any POWER or PowerPC processor. GCC will use only the instructions in the common subset of both architectures, and will not use the MQ register. GCC assumes a generic processor model for scheduling purposes.

The `-mcpu` options automatically enable or disable the `-maltivec`, `-mfprnd`, `-mhard-float`, `-mmfcrf`, `-mmultiple`, `-mnew-mnemonics`, `-mpopcntb`, `-mpower`, `-mpower2`, `-mpowerpc64`, `-mpowerpc-gpopt`, `-mpowerpc-gfxopt`, `-mstring`, `-mmulhw`, and `-mdlmzb` options to generate the best code possible for a specific processor. You can disable options explicitly if you are sure that you do not want to use them with your processor.

■ **Tip** On AIX systems, the `-maltivec` and `-mpowerpc64` options are not automatically enabled because they are not completely supported on the AIX platform. You can enable them manually if you are sure that they will work in your execution environment.

`-mdlmzb`: Specifying this option tells GCC to generate code that uses the string search dlmzb instructions, as needed. This option is active by default when compiling for the PowerPC 405 and 440 processors. (This option is not available in the GCC 4.x compilers.)

`-mdynamic-no-pic`: Specifying this option tells GCC to generate code that itself is not relocatable but in which all external references are relocatable. The resulting code is suitable for use in applications but not in shared libraries. This option is used on Darwin and Mac OS X systems.

`-meabi`: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to adhere to the EABI, which is a set of modifications to the System V.4 specifications. This means that the stack is aligned to an 8-byte boundary, a function `__eabi` is called to from `main()` to set up the EABI environment, and the `-msdata` option can use both `r2` and `r13` to point to two separate small data areas. This option is the default if you configure GCC using one of the `powerpc*-*-eabi*` build targets.

`-memb`: Specifying this option on embedded PowerPC systems tells GCC to set the `PPC_EMB` bit in the ELF flags header to indicate that EABI-extended relocations are used.

`-mfloat-gprs` | `-mfloat-gprs=`*`yes/single/double/no`*: Specifying either of these options enables GCC to generate floating-point operations on the general-purpose registers for architectures that support them. (This option is currently only supported on the MPC854x processors.) You can optionally specify an argument to this option to refine the type of floating-point operations that can be generated: `yes` and `single` enable the use of single-precision floating-point operations; `double` enables the use of both single and double-precision floating-point operations, and `no` disables floating-point operations on general-purpose registers.

`-mfprnd`: Specifying this option enables GCC to generate the floating-point round to integer instructions that are implemented on the POWER5+ and other PowerPC v2.03-compliant processors.

`-mfull-toc`: Specifying this option modifies the generation of the table of contents (TOC) generated for every GCC executable. If the `-mfull-toc` option is specified, GCC allocates at least one TOC entry for each unique nonautomatic variable reference in a program and will also place floating-point constants in the TOC. A maximum of 16,384 entries are available in the TOC. The `-mfull-toc` option is the default.

`-mfused-madd`: Specifying this option tells GCC to generate code that uses the floating-point multiply and accumulate instructions. These instructions are generated by default if hardware floating point is used.

`-mhard-float`: Specifying this option tells GCC to generate code that uses the floating-point register set.

`-minsert-sched-nops=`*`scheme`*: Specifying this option identifies the *scheme* that GCC will use for inserting NOOPs during its second scheduling pass. Possible values for *`scheme`* are `NO` (doesn't insert NOOPs), *`NUMBER`* (inserts *`NUMBER`* NOOPs to force costly dependent instructions into separate groups), *`PAD`* (pads any dispatch groups that has vacant slots with NOOPs), and *`REGROUP-EXACT`* (inserts NOOPs to force costly dependent instructions into separate groups, based on the grouping for the target processor).

`-misel`: Specifying this option enables GCC to generate `isel` instructions. This option replaces the `-misel=YES` syntax.

`-mlittle` | `-mlittle-endian`: Specifying either of these options on SVR4 or embedded PowerPC systems tells GCC to compile code for the processor in little endian mode.

`--mlongcall`: Specifying this option tells GCC to translate direct calls to indirect calls unless it can determine that the target of a direct call is in the 32MB range allowed by the call instruction. This can generate slower code on systems whose linker can automatically generate and insert glue code for out-of-range calls, such as the AIX, Darwin, and PowerPC-64 linkers, though the Darwin discards this code if it is unnecessary.

`-mmfcrf`: Specifying this option enables GCC to generate the move from condition register field instructions that are implemented on POWER4 and other PowerPC v2.01-compliant processors.

`-mminimal-toc`: Specifying this option modifies the generation of the TOC that is generated for every PPC executable. The TOC provides a convenient way of looking up the address/entry point of specific functions. This option is a last resort if you see a linker error indicating that you have overflowed the TOC during final linking and have already tried using the `-no-fp-in-toc` and `-mno-sum-in-toc` options. The `-mminimal-toc` option causes GCC to make only one TOC entry for every file. When you specify this option, GCC produces code that is slower and larger but uses extremely little TOC space. You may wish to use this option only on files that contain code that is infrequently executed.

`-mmulhw`: Specifying this option tells GCC to generate code that uses the half-word multiply and multiply-accumulate instructions, as needed. This option is enabled by default when compiling for the PowerPC 405 and 440 processors.

`-mmultiple`: Specifying this option tells GCC to generate code that uses the load multiple word instructions and the store multiple word instructions. These instructions are generated by default on POWER systems, and not generated on PowerPC systems. Do not use `-mmultiple` on little endian PowerPC systems except for the PPC740 and PPC750, since those instructions do not usually work when the processor is in little endian mode. The PPC740 and the PPC750 permit the use of these instructions in little endian mode.

`-mmvme`: Specifying this option on embedded PowerPC systems tells GCC to assume that the startup module is called crt0.o and the standard C libraries are libmvme.a and libc.a.

`-mnew-mnemonics`: Specifying this option tells GCC to use the assembler mnemonics defined for the PowerPC architecture. Instructions defined in only one architecture have only one mnemonic. GCC uses that mnemonic irrespective of which of these options is specified. GCC defaults to the mnemonics appropriate for the architecture that is in use. Unless you are cross-compiling, you should generally accept the default.

`-mno-altivec`: Specifying this option tells GCC to disable the use of built-in functions that provide access to the AltiVec instruction set.

`-mno-bit-align`: Specifying this option on SVR4 and embedded PowerPC systems tells GCC not to force structures and unions that contain bit fields to be aligned to the base type of the bit field. For example, by default, a structure containing nothing but eight unsigned bit fields of length 1 would be aligned to a 4-byte boundary and have a size of 4 bytes. When using the `-mno-bit-align` option, the structure would be aligned to a 1-byte boundary and be 1 byte in size.

`-mno-dlmzb`: Specifying this option prevents GCC from generating code that uses the string search dlmzb instructions that are otherwise used by default when compiling for the PowerPC 405 and 440 processors.

`-mno-eabi`: Specifying this option on SVR4 and embedded PowerPC systems tells GCC not to adhere to the EABI, which is a set of modifications to the System V.4 specifications. This means that the stack is aligned to a 16-byte boundary, an initialization function is not called from `main`, and that the `-msdata` option will only use r13 to point to a single small data area. This is the default for all GCC build configurations other than the `powerpc*-*-eabi*` build targets.

`-mno-fp-in-toc`: Specifying this option tells GCC to generate the same TOC as specified by the `-mfull-toc` option, but not to store floating-point constants in the TOC. This option and the `-mno-sum-in-toc` option are typically used if you see a linker error indicating that you have overflowed the TOC during final linking.

`-mno-fprnd`: Specifying this option prevents GCC from generating the floating-point round to integer instructions that are implemented on the POWER5+ and other PowerPC V2.03-compliant processors.

`-mno-fused-madd`: Specifying this option tells GCC to generate code that does not use the floating-point multiply and accumulate instructions. These instructions are generated by default if hardware floating point is used.

`-mno-isel`: Specifying this option prevents GCC from generating `isel` instructions. This option replaces the `-misel=NO` syntax.

`-mno-longcalls`: Specifying this option tells GCC not to translate direct calls to indirect calls.

`-mno-mfcrf`: Specifying this option prevents GCC from generating the move from condition register field instructions that are implemented on POWER4 and other PowerPC v2.01-compliant processors.

`-mno-mulhw`: Specifying this option tells GCC to generate code that does not use the half-word multiply and multiply-accumulate instructions that are otherwise generated by default when compiling for the PowerPC 405 and 440 processors.

`-mno-multiple`: Specifying this option tells GCC to generate code that does not use the load multiple word instructions or the store multiple word instructions. This option should not be used on little endian systems, with the exception of the 740 and 750 systems. These instructions are generated by default on POWER systems, and are not generated on PowerPC systems.

`-mno-popcntb`: Specifying this option enables GCC to generate the popcount and double-precision floating-point reciprocal estimate instructions that are implemented on the POWER5 and other PowerPC v20.20-compliant processors.

`-mno-power`: Specifying this option prevents GCC from generating code that uses any of the instructions that are specific to the POWER architecture.

■ **Note** If you specify both the `-mno-power` and `-mno-powerpc` options, GCC will use only the instructions in the common subset of both architectures plus some special AIX common-mode calls, and will not use the MQ register.

`-mno-power2`: Specifying this option prevents GCC from generating code that uses any of the instructions that are specific to the POWER2 architecture.

`-mno-powerpc`: Specifying this option prevents GCC from generating code that uses any of the instructions that are specific to the PowerPC architecture.

■ **Note** If you specify both the `-mno-power` and `-mno-powerpc` options, GCC will use only the instructions in the common subset of both architectures plus some special AIX common-mode calls, and will not use the MQ register.

`-mno-powerpc64`: Specifying this option prevents GCC from generating code that uses any of the instructions that are specific to the PowerPC-64 architecture.

`-mno-powerpc-gpopt`: Specifying this option prevents GCC from generating code that uses any of the instructions that are specific to the PowerPC architecture, including floating-point square root and the optional PowerPC architecture instructions in the general purpose group.

`-mno-powerpc-gfxopt`: Specifying this option prevents GCC from generating code that uses any of the instructions that are specific to the PowerPC architecture, including floating-point select and the optional PowerPC architecture instructions in the Graphics group.

`-mno-prototype`: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to assume that all calls to variable argument functions are properly prototyped. Only calls to prototyped variable argument functions will set or clear bit 6 of the condition code register (CR) to indicate whether floating-point values were passed in the floating-point registers.

`-mno-regnames`: Specifying this option on SVR4 and embedded PowerPC systems tells GCC not to emit register names in the assembly language output using symbolic forms.

`-mno-relocatable`: Specifying this option on embedded PowerPC systems tells GCC not to generate code that enables the program to be relocated to a different address at runtime. This minimizes the size of the resulting executable, such as a boot monitor or kernel.

`-mno-relocatable-lib`: Specifying this option on SVR4 and embedded PowerPC systems tells GCC not to generate code that enables the program to be relocated to a different address at runtime.

`-mno-secure-plt`: Specifying this option prevents GCC from generating code that allows the linker to build shared libraries with nonexecutable .plt and .got sections.

`-mno-spe`: Specifying this option prevents GCC from generating code that uses the SPE simd instructions. This option replaces the `-mspe=NO` syntax.

`-mno-strict-align`: Specifying this option on SVR4 and embedded PowerPC systems tells GCC not to assume that unaligned memory references will be handled by the system.

`-mno-string`: Specifying this option tells GCC to generate code that does not use the load string instructions or the store string word instructions. These instructions are generated by default on POWER systems, and not generated on PowerPC systems.

`-mno-sum-in-toc`: Specifying this option tells GCC to generate the same TOC as specified by the `-mfull-toc` option, but to generate code to calculate the sum of an address and a constant at runtime instead of putting that sum into the TOC. This option and the `-no-fp-in-toc` option are typically used if you see a linker error indicating that you have overflowed the TOC during final linking.

`-mno-swdiv`: Specifying this option prevents GCC from generating code that performs division in software.

`-mno-toc`: Specifying this option on SVR4 and embedded PowerPC systems tells GCC not to assume that register `2` contains a pointer to a global area pointing to the addresses used in the program.

`-mno-update`: Specifying this option tells GCC to generate code that does not use the load or store instructions that update the base register to the address of the calculated memory location. (These instructions are generated by default.) If you use the `-mno-updateg` option, there is a small window between the time that the stack pointer is updated and when the address of the previous frame is stored, which means that code that walks the stack frame across interrupts or signals may get corrupted data.

`-mno-vrsave`: Specifying this option prevents GCC from generating vrsave instructions when generating AltiVec code.

`-mno-xl-compat`: Specifying this option prevents GCC from performing the code gymnastics necessary to conform to the calling conventions of the IBM XL compilers on AIX systems. This is the default.

`-mold-mnemonics`: Specifying this option tells GCC to use the assembler mnemonics defined for the POWER architecture. Instructions defined in only one architecture have only one mnemonic. GCC uses that mnemonic irrespective of which of these options is specified. GCC defaults to the mnemonics appropriate for the architecture that is in use. Unless you are cross-compiling, you should generally accept the default.

`-mpe`: Specifying this option tells GCC to support the IBM RS/6000 SP Parallel Environment (PE). Applications written to use message passing must be linked with special startup code to enable the application to run. The system must have PE installed in the standard location (`/usr/lpp/ppe.poe/`), or GCC's specs file must be overridden by using the `-specs=` option to specify the appropriate directory location. The Parallel Environment does not support threads, so the `-mpe` option and the `-pthread` option are incompatible.

`-mpopcntb`: Specifying this option enables GCC to generate the popcount and double-precision floating-point reciprocal estimate instructions that are implemented on the POWER5 and other PowerPC v20.20-compliant processors.

`-mprioritize-restricted-insns=`*`priority`*: Specifying this option controls the priority assigned to dispatch-slot restricted instructions during GCC's second scheduling pass. Possible values for *`priority`* are `0` (no priority), `1` (highest priority), and `2` (second-highest priority).

`-mpower`: Specifying this option tells GCC to generate code using instructions that are found only in the POWER architecture and to use the MQ register.

Note Specifying both of the `-mpower` and `-mpowerpc` options permits GCC to use any instruction from either architecture and to allow use of the MQ register. Both of these options should be specified when generating code for the Motorola MPC601 processor.

`-mpower2`: Specifying this option implies the `-mpower` option and also enables GCC to generate instructions that are present in the POWER2 architecture but not the original POWER architecture.

`-mpowerpc`: Specifying this option tells GCC to generate instructions that are found only in the 32-bit subset of the PowerPC architecture.

Note Specifying both of the `-mpowerpc` and `-mpower` options permits GCC to use any instruction from either architecture and to allow use of the MQ register. Both of these options should be specified when generating code for the Motorola MPC601 processor.

`-mpowerpc64`: Specifying this option tells GCC to generate any PowerPC instructions as well as the additional 64-bit instructions that are found in the full PowerPC-64 architecture and to treat GPRs (general purpose registers) as 64-bit, double-word quantities. GCC defaults to `-mno-powerpc64`.

`-mpowerpc-gfxopt`: Specifying this option implies the `-mpowerpc` option and also enables GCC to use the optional PowerPC architecture instructions in the Graphics group, including floating-point select.

`-mpowerpc-gpopt`: Specifying this option implies the `-mpowerpc` option and also enables GCC to use the optional PowerPC architecture instructions in the General Purpose group, including floating-point square root.

`-mprototype`: Specifying this option on SVR4 and embedded PowerPC systems tells GCC not to assume that all calls to variable argument functions are properly prototyped. The compiler will insert an instruction before every nonprototyped call to set or clear bit 6 of the CR to indicate whether floating-point values were passed in the floating-point registers in case the function takes a variable number of arguments.

`-mregnames`: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to emit register names in the assembly language output using symbolic forms.

`-mrelocatable`: Specifying this option on embedded PowerPC systems tells GCC to generate code that enables the program to be relocated to a different address at runtime. If you use the `-mrelocatable` option on any module, all objects linked together must be compiled with the `-mrelocatable` or `-mrelocatable-lib` options.

`-mrelocatable-lib`: Specifying this option on embedded PowerPC systems tells GCC to generate code that enables the program to be relocated to a different address at runtime. Modules compiled with the `-mrelocatable-lib` option can be linked with modules compiled without the `-mrelocatable` option.

`-msched-costly-dep=`*`dependence-type`*: Specifying this option identifies the dependencies that are considered costly during instruction scheduling. Possible values for *`dependence-type`* are `NO` (no dependencies are especially costly), `ALL` (all dependencies are costly), `NUMBER` (any dependency with latency greater than `NUMBER` is costly), `STORE_TO_LOAD` (any dependency from store to load is costly), and `TRUE_STORE_TO_LOAD` (a true dependency from store to load is costly).

`-msdata=abi`: Specifying this option tells GCC where to put small global and static data based on the rules for various platforms. Possible values for *`abi`* are `default` (behaves as if the `-msdata=sysv` option were specified, unless the `-meabi` option was also specified, in which case it behaves as if the `-msdata=eabi` option were specified), `eabi` (puts small initialized global and static data in the sdata2 section, puts small initialized non-const global and static data in the sdata section pointed to by register `13`, and puts small uninitialized global and static data in the sbss section adjacent to the stat section), `none` (puts all initialized global and static data in the data section and all uninitialized data in the bss section), and `sysv` (puts small global and static data in the sdata section, which is pointed to by register `r13`, and small uninitialized global and static data in the sbss section adjacent to the sdata section). The `-msdata=none` option can also be specified as `-mno-sdata`. The `-msdata=eabi` and `-msdata=sysv` options are incompatible with the `-mrelocatable` option. The `-msdata=eabi` option also sets the `-memb` option.

`-msdata-data`: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to put small global and static data in the sdata section. Small uninitialized global and static data are put in the sbss section, but register `r13` is not used to address small data. This is the default behavior unless other `-msdata` options are specified.

`-msecure-plt`: Specifying this option enables GCC to generate code that allows the linker to build shared libraries with nonexecutable .plt and .got sections. This is a PowerPC 32-bit SYSV ABI option.

`-msim`: Specifying this option on embedded PowerPC systems tells GCC to assume that the startup module is called sim-crt0.o and that the standard C libraries are libsim.a and libc.a. This is the default for the `powerpc-*-eabisim` GCC build configuration.

`-msoft-float`: Specifying this option tells GCC to generate code that does not use the floating-point register set. Software floating-point emulation is provided if you use this option and pass it to GCC when linking.

`-mspe`: Specifying this option tells GCC to generate code that uses the SPE simd instructions, as needed. This option replaces the `-mspe=YES` syntax.

`-mstrict-align`: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to assume that unaligned memory references will be handled by the system.

`-mstring`: Specifying this option tells GCC to generate code that uses the load string instructions and the store string word instructions to save multiple registers and do small block moves. These instructions are generated by default on POWER systems, and not generated on PowerPC systems. Do not use `-mstring` on little endian PowerPC systems except for PPC740 and PPC750 systems, since those instructions usually do not work when the processor is in little endian mode. PPC740 and PPC750 permit the use of these instructions in little endian mode.

`-msvr4-struct-return`: Specifying this option tells GCC to return structures smaller than 8 bytes in registers, as specified by the SVR4 ABI.

`-mswdiv`: Specifying this option tells GCC to generate code that performs division in software using reciprocal estimates and iterative refinement. This can provide increased throughput, but requires the PowerPC graphics instruction set for single precision calculations and the FRE instruction for double-precision calculations, assumes that division calculations cannot generate user-visible traps, and assumes that possible values can not include infinite values, denormals, or zero denominators.

`-mtoc`: Specifying this option tells GCC to assume that register `2` contains a pointer to a global area pointing to the addresses used in the program on SVR4 and embedded PowerPC systems.

`-mtune=`*`cpu-type`*: Specifying this option tells GCC to set the instruction scheduling parameters for the machine type specified by *`cpu-type`*, but not to set the architecture type, register usage, or choice of mnemonics, as `-mcpu=`*`cpu-type`* would. The same values for *`cpu-type`* are used for both the `-mtune` and `-mcpu` options. If both are specified, the code generated will use the architecture, registers, and mnemonics set by the argument to `-mcpu`, but the scheduling parameters set by the argument to `-mtune`.

`-mupdate`: Specifying this option tells GCC to generate code that uses the load or store instructions that update the base register to the address of the calculated memory location. These instructions are generated by default.

`-mvrsave`: Specifying this option tells GCC to generate vrsave instructions when generating AltiVec code.

`-mvxworks`: Specifying this option on SVR4 and embedded PowerPC systems tells GCC that you are compiling for a VxWorks system. You have my sympathy.

`-mwindiss`: Specifying this option tells GCC that you are compiling for the WindISS simulation environment.

`-mxl-compat`: Specifying this option when compiling using an AIX-compatible ABI tells GCC to pass floating-point arguments to prototyped functions beyond the register save area (RSA) on the stack in addition to argument floating-point registers. When a subroutine is compiled without optimization, IBM AIX XL compilers access floating-point arguments that do not fit in the RSA from the stack. Because always storing floating-point arguments on the stack is inefficient and rarely needed, this option is not enabled by default and is only necessary when calling subroutines compiled without optimization by IBM AIX XL compilers.

`-myellowknife`: Specifying this option on embedded PowerPC systems tells GCC to assume that the startup module is called crt0.o and the standard C libraries are libyk.a and libc.a. *Yellowknife* is the name of an old Motorola evaluation board.

`-pthread`: Specifying this option tells GCC to add support for multithreading through the pthreads library. This option sets flags for both the preprocessor and the linker. (This option is not available in the GCC 4.x compilers.)

RS/6000 Options

See the section "PowerPC (PPC) Options" of this appendix for options relevant to using GCC to compile applications for IBM's RS/6000 family of workstations.

RT Options

The IBM RT was IBM's first RISC workstation, and saw little use outside academia. It originally ran an operating system called Academic Operating System (AOS), which was loosely based on BSD Unix, but was later the original deployment platform for IBM's AIX operating system. The IBM RT was also a primary development platform for the Mach operating system at Carnegie Mellon University.

■ **Note** Support for this processor family is not provided in GCC version 4.0. These options are therefore only of interest if you are using a version of GCC that is earlier than 4.x and that you are certain provides support for this processor.

GCC options available when compiling code for IBM RT systems are the following:

`-mcall-lib-mul`: Specifying this option tells GCC to call `lmul$$` for integer multiples.

`-mfp-arg-in-fpregs`: Specifying this option tells GCC to use a calling sequence in which floating-point arguments are passed in floating-point registers. This calling sequence is incompatible with the IBM calling convention. Note that varargs.h and stdarg.h will not work with floating-point operands if this option is specified.

`-mfp-arg-in-gregs`: Specifying this option tells GCC to use the normal calling convention for floating-point arguments. This is the default.

`-mfull-fp-blocks`: Specifying this option tells GCC to generate full-size floating-point data blocks, including the minimum amount of scratch space recommended by IBM. This is the default.

`-mhc-struct-return`: Specifying this option tells GCC to return structures of more than one word in memory, rather than in a register. This provides compatibility with the MetaWare HighC compiler. See the `-fpcc-struct-return` option for similar compatibility with the Portable C Compiler (PCC).

`-min-line-mul`: Specifying this option tells GCC to use an inline code sequence for integer multiplies. This is the default.

`-mminimum-fp-blocks`: Specifying this option tells GCC not to include extra scratch space in floating-point data blocks. This results in smaller code but slower execution, since scratch space must be allocated dynamically.

`-mnohc-struct-return`: Specifying this option tells GCC to return some structures of more than one word in registers, when convenient. This is the default.

`-mpcc-struct-return`: Specifying this option tells GCC to use return conventions that are compatible with PCC.

S/390 and zSeries Options

The S/390 is the latest in IBM's family of 360 and 370 systems that stretches back to the 1960s. The zSeries is IBM's name for its latest series of high-powered servers. S/390 systems can run Linux in virtual machines, just as they run other operating systems. The zSeries machines can run Linux directly.

Linux supports the S/390 and zSeries processor architecture and some devices that are specific to S/390 and zSeries environments. Linux on these systems enables you to take advantage of the fast I/O and reliability that are traditional features of S/390 and zSeries mainframe hardware.

GCC options available when compiling code for S/390 and zSeries systems are the following:

`-m31`: Specifying this option tells GCC to generate code that is compliant to the Linux for S/390 ABI. This is the default for s390 targets.

`-m64`: Specifying this option tells GCC to generate code that is compliant to the Linux for zSeries ABI. This allows GCC to generate 64-bit instructions. This is the default for s390x build targets.

`-march=`*`CPU-type`*: Specifying this option tells GCC the instruction set to use for code generation. Possible values for *`CPU-type`* are `g5`, `g6`, `z900`, and `z990`. When generating code using the z/Architecture instructions, the default value for *`CPU-type`* is `z900`. Otherwise, the default value for `CPU-type` is `g5`.

`-mbackchain`: Specifying this option tells GCC to generate code that maintains an explicit backchain within the stack frame. This backchain points to the caller's frame, and is currently needed to allow debugging. This is the default.

`-mdebug`: Specifying this option tells GCC to print additional debug information when compiling.

`-mesa`: Specifying this option tells GCC to generate code using only the instructions that are available on the ESA/90. This option is the default when generating Linux code for the ESA/90 ABI.

`-mfused-madd`: Specifying this option tells GCC to generate code that uses the floating-point multiply and accumulate instructions when they are available. These instructions are generated by default if they are available.

`-mguard-size=`*`GUARD-SIZE`*: See the `-mstack-size` option for more information.

`-mhard-float`: Specifying this option tells GCC to use the hardware floating-point instructions and registers for floating-point operations. GCC generates the appropriate IEEE floating-point instructions. This is the default.

`-mlong-double-64`: Specifying this option tells GCC to use the default size of 64 bits for both the `long double` and `double` datatypes. This is the default.

`-mlong-double-128`: Specifying this option tells GCC to double the size of the `long double` datatype to 128 bits. By default it is 64 bits and is therefore internally equivalent to the `double` datatype.

`-mmvcle`: Specifying this option tells GCC to generate code using the mvcle instruction to perform block moves.

`-mno-backchain`: Specifying this option tells GCC not to generate code that maintains an explicit backchain within the stack frame that points to the caller's frame. Not maintaining a backchain prevents debugging.

`-mno-debug`: Specifying this option tells GCC not to print additional debug information when compiling. This is the default.

`-mno-fused-madd`: Specifying this option prevents GCC from generating code that uses the floating-point multiply and accumulate instructions, even when they are available.

`-mno-mvcle`: Specifying this option tells GCC not to generate code using the mvcle instruction to perform block moves, but to use an `mvc` loop instead. This is the default.

`-mno-packed-stack`: Specifying this option tells GCC not to use the packed stack layout, and to therefore only use fields in the register save area for their default purposes—unused space in this area is wasted. Code generated with this option is compatible with code that is not generated with this option as long as the `-mbackchain` option is not used.

`-mno-small-exec`: Specifying this option tells GCC not to generate code using the bras instruction to do subroutine calls. This is the default, causing programs compiled with GCC to use the basr instruction instead, which does not have a 64K limitation.

`-mno-tpf-trace`: Specifying this option prevents GCC from generating code that adds 390/TPF (Transaction Processing Facility) branches to trace routines in that operating system. This is the default, even when compiling for 390/TPF.

`-mpacked-stack`: Specifying this option tells GCC to use the packed stack layout where register save slots are densely packed so that unused space can be used for other purposes. Code generated with this option is compatible with code that is not generated with this option as long as the `-mbackchain` option is not used.

`-msmall-exec`: Specifying this option tells GCC to generate code using the bras instruction to do subroutine calls. This only works reliably if the total executable size does not exceed 64K.

`-msoft-float`: Specifying this option tells GCC not to use the hardware floating-point instructions and registers for floating-point operations. Functions in libgcc.a will be used to perform floating-point operations.

`-mstack-size=`*`stack-size`*: Specifying this option causes GCC to emit code in the function prologue that triggers a trap if the stack size is a specified number of bytes above `stack-size`. This option must be used with the `-mstack-guard=`*`guard-size`* option, whose `guard-size` argument identifies the number of bytes above `stack-size` at which a trap will be triggered. The value specified for *`stack-size`* must be greater than *`guard-size`* and not exceed 64K. Both `stack-size` and `guard-size` values must be powers of 2.

`-mtpf-trace`: Specifying this option tells GCC to generate code that adds 390/TPF (Transaction Processing Facility) branches to trace routines in that operating system. This option is off by default, even when compiling for 390/TPF.

`-mvcle`: Specifying this option tells GCC to generate using the mvcle instruction to perform block moves, as needed. This option is off by default.

`-mwarn-dynamicstack`: Specifying this option causes GCC to emit a warning if a function calls alloca or uses dynamically sized arrays. This option can be useful when compiling any executables that have limited stack size, such as the Linux kernel.

`-mwarn-framesize=`*`FRAMESIZE`*: Specifying this option causes GCC to emit a warning if a function exceeds the specified *`FRAMESIZE`*, and is intended to help identify function during compilation that might cause a stack overflow. This option can be useful when compiling any executables that have limited stack size, such as the Linux kernel.

`-mzarch`: Specifying this option tells GCC to generate code using only the instructions that are available on the z/Architecture. This option is the default when generating Linux code for the zSeries ABI.

SH Options

SuperH (SH) processors from Hitachi, Renesas, and others are powerful and popular processors for use in embedded systems. The SH1 and SH2 processors are 16-bit processors, the SH3, SH4, and SH4a are 32-bit processors, and the new SH5 is a fast 64-bit processor.

GCC options available when compiling code for systems using SH processors are the following:

`-m1`: Specifying this option tells GCC to generate code for the SH1 processor.

`-m2`: Specifying this option tells GCC to generate code for the SH2 processor.

`-m2e`: Specifying this option tells GCC to generate code for the SH2e processor.

`-m3`: Specifying this option tells GCC to generate code for the SH3 processor.

`-m3e`: Specifying this option tells GCC to generate code for the SH3e processor.

`-m4-nofpu`: Specifying this option tells GCC to generate code for SH4 processors without a floating-point unit.

`-m4-single-only`: Specifying this option tells GCC to generate code for SH4 processors with a floating-point unit that only supports single precision arithmetic.

`-m4-single`: Specifying this option tells GCC to generate code for SH4 processors, assuming the floating-point unit is in single precision mode by default.

`-m4`: Specifying this option tells GCC to generate code for the SH4 processor.

`-m4a-nofpu`: Specifying this option tells GCC to generate code for SH4a processors without a floating-point unit.

`-m4a-single-only`: Specifying this option tells GCC to generate code for SH4a processors with a floating-point unit that only supports single precision arithmetic.

`-m4a-single`: Specifying this option tells GCC to generate code for SH4a processors, assuming the floating-point unit is in single precision mode by default.

`-m4a`: Specifying this option tells GCC to generate code for the SH4a processor.

`-m4al`: Specifying this option tells GCC to generate code for SH4a1-DSP or for SH4a processors without a floating-point unit. This option is the same as the `-m4a-nofpu` except that it also specifies the `-dsp` option.

`-madjust-unroll`: Specifying this option tells GCC to limit loop unrolling in order to avoid thrashing on target registers. For this option to have any effect, the GCC code base must support the `TARGET_ADJUST_UNROLL_MAX` hook.

`-mb`: Specifying this option tells GCC to compile code for an SH processor in big endian mode.

`-mbigtable`: Specifying this option tells GCC to use 32-bit offsets in switch tables. The default is to use 16-bit offsets.

`-mdalign`: Specifying this option tells GCC to align `doubles` at 64-bit boundaries. This changes the calling conventions, and therefore some functions from the standard C library will not work unless you also recompile the standard C library with the `-mdalign` option.

`-mdiv=`*`strategy`*: Specifying this option tells GCC the division strategy that it should use when generating code using Super-Hitachi's SHmedia SIMD instruction set. Valid values for *`strategy`* are `call`, `call2`, `fp`, `inv`, `inv:minlat`, `inv20u`, `inv20l`, `inv:call`, `inv:call2`, and `inv:fp`.

`-mdivsi3_libfunc=`*name*: Specifying this option tells GCC the name of the library to use for 32-bit signed division. This only affects the actual function calls; GCC expects the same behavior and register-use conventions as if this option was not specified.

`-mfmovd`: Specifying this option tells GCC to enable the use of the fmovd instruction.

`-mgettrcost=`*number*: Specifying this option tells GCC the cost (*number*) to assume when scheduling the gettr instruction. The default value for *number* is `2` if the `-mpt-fixed` option is specified, and `100` otherwise.

`-mhitachi`: Specifying this option tells GCC to comply with the SH calling conventions defined by Hitachi.

`-mieee`: Specifying this option tells GCC to generate IEEE-compliant floating-point code.

`-mindexed-addressing`: Specifying this option enables GCC to use the indexed addressing mode for SHmedia32/SHcompact, which enables the implementation of 64-bit MMUs but is only safe when the hardware and/or OS implements 32-bit wraparound semantics, which is not currently supported on any hardware implementation.

`-minvalid-symbols`: Specifying this option tells GCC to assume that symbols may be invalid due to cross-basic-block common subexpression elimination or hoisting.

`-misize`: Specifying this option tells GCC to include instruction size and location in the assembly code.

`-ml`: Specifying this option tells GCC to compile code for an SH processor in little endian mode.

`-no-minvalid-symbols`: Specifying this option tells GCC to assume that all symbols are valid. This option is the default.

`-mno-pt-fixed`: Specifying this option tells GCC to assume that pt* instructions may generate a trap.

`-mno-renasas`: Specifying this option tells GCC to comply with the SH calling conventions used before the Renesas calling conventions were available. This option is the default for all SH toolchain targets except for sh-symbianelf.

`-mnomacsave`: Specifying this option tells GCC to mark the MAC register as call-clobbered, even if the `-mhitachi` option is also used.

`-mpadstruct`: This is a deprecated option that pads structures to multiples of 4 bytes, which is incompatible with the SH ABI.

`-mprefergot`: Specifying this option tells GCC to emit function calls using the global offset table instead of the procedure linkage table when generating position-independent code.

`-mpt-fixed`: Specifying this option tells GCC that no pt* instructions will generate a trap. This improves scheduling but is unsafe on current hardware.

`-mrelax`: Specifying this option tells GCC to shorten some address references at link time, when possible. Specifying this option passes the `-relax` option to the linker.

`-mrenasas`: Specifying this option tells GCC to comply with the SH calling conventions defined by Renesas.

`-mspace`: Specifying this option tells GCC to optimize for size instead of speed. This option is implied by the `-Os` optimization option.

`-multcost=`*`number`*: Specifying this option tells GCC the cost (an integer *number*) to assume for a multiply instruction when scheduling.

`-musermode`: Specifying this option tells GCC to generate a library function call that invalidates instruction cache entries after fixing up a trampoline. The function call does not assume that it can write to the whole memory address space. This is the default when GCC was built for the sh-*-linux* target.

SPARC Options

The SPARC is a fast 32-bit processor architecture originally developed by Sun Microsystems and used in all of its workstations, single-board computers (SBCs), and other embedded hardware since the late 1980s. The SPARC processor has also been licensed to a number of other workstation and embedded hardware vendors. The latest generation of SPARC processors, the UltraSPARC family, are 64-bit processors.

GCC options available when compiling code for systems using SPARC processors are the following:

`-m32`: Specifying this option tells GCC to generate code for a 32-bit environment. The 32-bit environment sets `int`, `long`, and `pointer` to 32 bits. This option is only supported when compiling for SPARC V9 processors in 64-bit environments.

`-m64`: Specifying this option tells GCC to generate code for a 64-bit environment. The 64-bit environment sets `int` to 32 bits and `long` and `pointer` to 64 bits.

`-mapp-regs`: Specifying this option tells GCC to be fully SVR4 ABI compliant at the cost of some performance loss. Libraries and system software should be compiled with this option to maximize compatibility.

`-mcmodel=`*`code-model`*: Specifying this option identifies the code model that GCC should use when generating code. Valid values for *code-model* are `embmedany`, `medany`, `medlow`, and `medmid`. See the explanations of each of these option/*`code-model`* pairs for detailed information about the code model that they implement.

`-mcmodel=embmedany`: Specifying this option tells GCC to generate code for the Medium/Anywhere code model for embedded systems, which assumes a 32-bit text and a 32-bit data segment, both starting anywhere (determined at link time). Register `%g4` points to the base of the data segment. Pointers are still 64 bits. Programs are statically linked; PIC is not supported. This option is only supported when compiling for SPARC V9 processors in 64-bit environments.

`-mcmodel=medany`: Specifying this option tells GCC to generate code for the Medium/Anywhere code model, where the program may be linked anywhere in the address space, the text segment must be less than 2GB, and the data segment must be within 2GB of the text segment. Pointers are 64 bits. This option is only supported when compiling for SPARC V9 processors in 64-bit environments.

`-mcmodel=medlow`: Specifying this option tells GCC to generate code for the Medium/Low code model, where a program must be linked in the low 32 bits of the address space. Pointers are 64 bits. Programs can be statically or dynamically linked. This option is only supported when compiling for SPARC V9 processors in 64-bit environments.

`-mcmodel=medmid`: Specifying this option tells GCC to generate code for the Medium/Middle code model, where the program must be linked in the low 44 bits of the address space, the text segment must be less than 2GB, and the data segment must be within 2GB of the text segment. Pointers are 64 bits. This option is only supported when compiling for SPARC V9 processors in 64-bit environments.

`-mcpu=CPU-type`: Specifying this option tells GCC to set the instruction set, register set, and instruction scheduling parameters for machine type `CPU-type`. Supported values for *`CPU-type`* are `cypress`, `f930`, `f934`, `hypersparc`, `niagra`, `sparclet`, `sparclite`, `sparclite86x`, `supersparc`, `tsc701`, `ultrasparc`, `ultrasparc3`, `v6`, `v7`, `v8`, and `v9`. Default instruction scheduling parameters are used for values that select an architecture and not an implementation. These are `sparclet`, `sparclite`, `v7`, `v8`, and `v9`. By default, GCC generates code for the `v7` `CPU-type`.

Table B-1 shows each supported architecture and its supported implementations.

Table B-1. *SPARC Architectures and Associated CPU-type Values*

Values	Architecture
`v7`	cypress
`v8`	supersparc, hypersparc
`sparclite`	f930, f934, sparclite86x
`sparclet`	tsc701
`v9`	niagra, ultrasparc, ultrasparc3

`-mcypress`: Specifying this option (or the `-mcpu=cypress` option) tells GCC to optimize code for the Cypress CY7C602 chip, as used in the SPARCStation/SPARCServer 3xx series. This is also appropriate for the older SPARCStation 1, 2, IPX, and so on. This is the default. This option is deprecated—the more general `-mCPU=CPU-type` option should be used instead. (This option is not available in the GCC 4.x compilers.)

`-mfaster-structs`: Specifying this option tells GCC to assume that structures should have 8-byte alignment. This enables the use of pairs of ldd and std instructions for copies in structure assignment, instead of twice as many ld and st pairs. However, the use of this changed alignment directly violates the SPARC ABI and is therefore intended only for use on targets where developers acknowledge that their resulting code will not be directly in line with the rules of the ABI.

`-mflat`: Specifying this option tells GCC not to generate save/restore instructions and instead to use a flat, or single-register window, calling convention. This model uses `%i7` as the frame pointer and is compatible with code that does not use this calling convention. Regardless of the calling conventions used, the local registers and the input registers (`0–5`) are still treated as "call-saved" registers and will be saved on the stack as necessary.

`-mhard-float|-mfpu`: Specifying either of these options tells GCC to generate output containing floating-point instructions. This is the default.

`-mhard-quad-float`: Specifying this option tells GCC to generate output that contains quad-word (`long double`) floating-point instructions.

■ **Note** No current SPARC implementations provide hardware support for the quad-word floating-point instructions. They all invoke a trap handler for one of these instructions, where the trap handler then emulates the effect of the instruction. Because of the overhead of the trap handler, this is much slower than calling the ABI library routines. For this reason, the `-msoft-quad-float` option is the default.

`-mimpure-text`: Specifying this option along with the standard GCC `-shared` option tells GCC not to pass `-z text` to the linker when linking a shared object, which enables you to link position-independent code into shared objects. This option is only available when compiling for SunOS or Solaris.

`-mlittle-endian`: Specifying this option tells GCC to generate code for a processor running in little endian mode. This option is only supported when compiling for SPARC V9 processors in 64-bit environments and is not supported in most standard GCC configurations, such as Linux and Solaris.

`-mno-app-regs`: Specifying this option tells GCC to generate output using the global registers `2` through `4`, which the SPARC SVR4 ABI reserves for applications. This is the default.

`-mno-faster-structs`: Specifying this option tells GCC not to make any assumptions about structure alignment, and to use the ld and st instructions when making copies during structure assignment.

`-mno-flat`: Specifying this option tells GCC to use save/restore instructions as its calling convention (except for leaf functions). This option is the default.

`-mno-stack-bias`: Specifying this option tells GCC to assume that no offset is used when making stack frame references. This option is only supported when compiling for SPARC V9 processors in 64-bit environments.

`-mno-unaligned-doubles`: Specifying this option tells GCC to assume that `doubles` have 8-byte alignment. This is the default.

`-mno-v8plus`: Specifying this option tells GCC not to assume that in and out registers are 6 bits when generating code which essentially means that code generation conforms to the V8 ABI.

`-mno-vis`: Specifying this option prevents GCC from generating code that uses the UltraSPARC VIS (Visual Instruction Set) instructions. This is the default.

`-msoft-float` | `-mno-fpu`: Specifying either of these options tells GCC to generate output that contains library calls for floating-point operations. These libraries are not provided as part of GCC (except for the embedded GCC build targets sparc-*-aout and sparclite-*-*), but are normally found on the target system, and should be resolved by the C loader on the target machine if they are available. When using this option and cross-compiling, you must provide suitable libraries containing at least function stubs on the host system, or these function calls will be identified as unresolved.

■ **Tip** Specifying this option also changes the calling convention used in the output file. You must therefore compile all of the modules of your program with this option, including any libraries that you reference. You must also compile libgcc.a, the library that comes with GCC, with this option in order to be able to use this option.

`-msoft-quad-float`: Specifying this option tells GCC to generate output containing library calls for quad-word (`long double`) floating-point instructions. The functions called are those specified in the SPARC ABI. This is the default.

`-msparclite`: Specifying this option (or the `-mcpu=sparclite` option) tells GCC to generate code for SPARClite processors. This adds the integer multiply and integer divide step and scan (ffs) instructions that exist in SPARClite but not in SPARC V7. This option is deprecated; the more general `-mCPU=`*`cpu-type`* option should be used instead.

`-mstack-bias`: Specifying this option tells GCC to assume that the stack pointer and the frame pointer (if present) are offset by -2047, which must be added back when making stack frame references. This option is only supported when compiling for SPARC V9 processors in 64-bit environments.

`-msupersparc`: Specifying this option (or the `-mcpu=supersparc` option) tells GCC to optimize code for the SuperSPARC processor, as used in the SPARCStation 10, 1000, and 2000 series. This option also enables use of the full SPARC V8 instruction set. This option is deprecated; the more general `-mCPU=CPU-type` option should be used instead.

`-mtune=`*`cpu-type`*: Specifying this option tells GCC to set the instruction scheduling parameters for machine type *`cpu-type`*, but not to set the instruction set or register set as would the option `-mcpu=CPU-type`. The same values for `-mcpu=`*`cpu-type`* can be used with the `-mtune=`*`cpu-type`* option, but the only useful values are those that select a particular processor implementation: `cypress`, `f930`, `f934`, `hypersparc`, `sparclite86x`, `supersparc`, `tsc701`, and `ultrasparc`.

`-munaligned-doubles`: Specifying this option causes GCC not to assume that `doubles` are 8-byte aligned. GCC assumes that `doubles` have 8-byte alignment only if they are contained in another type, or if they have an absolute address. Otherwise, it assumes they have 4-byte alignment. Specifying this option avoids some rare compatibility problems with code generated by other compilers, but results in a performance loss, especially for floating-point code.

`-mv8plus`: Specifying this option tells GCC to generate code for the SPARC V8+ ABI. The only difference from the V8 ABI is that the global in and out registers are 64 bits wide. This option is enabled by default when compiling for SPARC V9 processors in 32-bit mode.

`-mvis`: Specifying this option tells GCC to generate code that uses the UltraSPARC VIS instructions, as needed.

`-pthread` | `-pthreads`: Specifying either of these options causes GCC to add support for multithreading using the POSIX thread library. Specifying this option sets other options for both the preprocessor and the linker.

`-threads`: Specifying this option causes GCC to add support for multithreading using the Solaris thread library. Specifying this option sets other options for both the preprocessor and the linker.

System V Options

System V Release 4, or SVR4, was a landmark release of the Unix operating system from AT&T and is the conceptual foundation for Unix-like operating systems such as Solaris and Linux. Alas, poor BSD, I knew thee well . . .

GCC options specific to compiling code for systems running SVR4 are the following:

`-G`: Specifying this option tells GCC to create a shared object. This option is deprecated, as it is easily confused with other `-G` options. You should use the more comprehensible `-symbolic` or `-shared` options instead.

`-Qn`: Specifying this option tells GCC not to add `.ident` directives identifying tool versions in the output assembly file. This is the default.

`-Qy`: Specifying this option tells GCC to identify the versions of each tool used by the compiler by adding an `.ident` assembler directive in the assembler output.

`-Ym,`*`dir`*: Specifying this option tells GCC to look in the directory *`dir`* to find the m4 preprocessor. The assembler uses this option.

`-YP,`*`dir`*: Specifying this option tells GCC to search the directory *`dir`*, and no others, for libraries specified with -l.

TMS320C3x/C4x Options

The TMS320C3x and TMS320C4x processors are digital signal processors from Texas Instruments.

GCC options specific to compiling code for systems running TMS320C3x and TMS320C4x processors are the following:

`-mbig` | `-mbig-memory`: Specifying either of these options tells GCC to generate code for the big memory model. Using the big memory model makes no assumptions about code data size and requires reloading of the `dp` register for every direct memory access. This is the default.

`-mbk`: Specifying this option tells GCC to allow allocation of general integer operands into the block count register `bk`.

`-mcpu=`*`cpu-type`*: Specifying this option tells GCC to set the instruction set, register set, and instruction scheduling parameters for machine type `CPU-type`. Supported values for `CPU-type` are `c30`, `c31`, `c32`, `c40`, and `c44`. The default is `c40`, which generates code for the TMS320C40.

`-mdb`: Specifying this option tells GCC to generate code that uses the decrement and branch, DBcond(D), instruction. This is the default for TMS320C4x processors.

Note GCC will try to reverse a loop so that it can utilize the decrement and branch instruction, but will give up if there is more than one memory reference in the loop. Thus a loop where the loop counter is decremented can generate slightly more efficient code in cases where the RPTB instruction cannot be utilized.

`-mdp-isr-reload` | `-mparanoid`: Specifying either of these options tells GCC to force the DP register to be saved on entry to an interrupt service routine (ISR), reloaded to point to the data section, and restored on exit from the ISR. This should not be necessary unless someone has violated the small memory model by modifying the DP register, such as within an object library.

`-mfast-fix`: Specifying this option tells GCC to accept the results of the C3x/C4x FIX instruction which, when converting a floating-point value to an integer value, chooses the nearest integer less than or equal to the floating-point value rather than to the nearest integer. Thus, if the floating-point number is negative, the result will be incorrectly truncated and additional code will be necessary to detect and correct this case.

`-mloop-unsigned`: Specifying this option tells GCC to use an unsigned iteration count. This limits loop iterations to 2**31 + 1, since these instructions test if the iteration count is negative in order to terminate a loop.

`-mmemparm`: Specifying this option tells GCC to generate code that uses the stack for passing arguments to functions.

`-mmpyi`: Specifying this option when compiling code for TMS320C3x processors tells GCC to use the 24-bit MPYI instruction for integer multiplies instead of a library call to guarantee 32-bit results. Note that if one of the operands is a constant, the multiplication will be performed using shifts and adds.

`-mno-bk`: Specifying this option tells GCC not to allow allocation of general integer operands into the block count register `bk`.

`-mno-db`: Specifying this option tells GCC not to generate code that uses the decrement and branch, DBcond(D), instruction. This is the default for TMS320C3x processors.

`-mno-fast-fix`: Specifying this option tells GCC to generate additional code to correct the results of the C3x/C4x FIX instruction which, when converting a floating-point value to an integer value, chooses the nearest integer less than or equal to the floating-point value rather than to the nearest integer. Thus, if the floating-point number is negative, the result will be incorrectly truncated. Specifying this option generates the additional code necessary to detect and correct this case.

`-mno-loop-unsigned`: Specifying this option tells GCC not to use an unsigned iteration count, which might artificially limit loop iterations to 2**31 + 1, since these instructions test if the iteration count is negative in order to terminate a loop.

`-mno-mpyi`: Specifying this option tells GCC to use a library call for integer multiplies. When compiling code for the TMS320C3x processor, squaring operations are performed inline instead of through a library call.

`-mno-parallel-insns`: Specifying this option tells GCC not to generate parallel instructions.

`-mno-parallel-mpy`: Specifying this option tells GCC not to generate MPY||ADD and MPY||SUB parallel instructions. This generally minimizes code size, though the lack of parallelism may negatively impact performance.

`-mno-rptb`: Specifying this option tells GCC not to generate repeat block sequences using the `RPTB` instruction for zero overhead looping.

`-mno-rpts`: Specifying this option tells GCC not to use the single-instruction repeat instruction RPTS. This is the default, because interrupts are disabled by the RPTS instruction.

`-mparallel-insns`: Specifying this option tells GCC to enable generating parallel instructions. This option is implied when the `-O2` optimization option is specified.

`-mparallel-mpy`: Specifying this option tells GCC to generate MPY||ADD and MPY||SUB parallel instructions if the `-mparallel-insns` option is also specified. These instructions have tight register constraints that can increase code size for large functions.

`-mregparm`: Specifying this option tells GCC to generate code that uses registers (whenever possible) for passing arguments to functions. This is the default.

`-mrptb`: Specifying this option tells GCC to enable the generation of repeat block sequences using the RPTB instruction for zero overhead looping. The RPTB construct is only used for innermost loops that do not call functions or jump across the loop boundaries. There is no advantage to having nested RPTB loops due to the overhead required to save and restore the `RC`, `RS`, and `RE` registers. This is option enabled by default with the `-O2` optimization option.

`-mrpts=`*`count`*: Specifying this option tells GCC to enable the use of the single instruction repeat instruction RPTS. If a repeat block contains a single instruction, and the loop count can be guaranteed to be less than the value *`count`*, GCC will emit an RPTS instruction instead of an RPTB instruction. If no value is specified, an RPTS will be emitted even if the loop count cannot be determined at compile time. Note that the repeated instruction following the RPTS instruction does not have to be reloaded from memory during each iteration, thus freeing up the CPU buses for operands.

`-msmall` | `-msmall-memory`: Specifying either of these options tells GCC to generate code for the small memory model. The small memory model assumes that all data fits into one 64K word page. At runtime, when using the small memory model, the DP register must be set to point to the 64K page containing the bss and data program sections.

`-mti`: Specifying this option tells GCC to try to emit an assembler syntax that the Texas Instruments assembler (asm30) is happy with. This also enforces compatibility with the ABI employed by the TI C3x C compiler. For example, `long doubles` are passed as structures rather than in floating-point registers.

V850 Options

NEC's V850 family of 32-bit RISC microcontrollers is designed for embedded real-time applications such as motor control, process control, industrial measuring equipment, automotive systems, and so on.

GCC options specific to compiling code for systems running V850 processors are the following:

`-mapp-regs`: Specifying this option tells GCC to use `r2` and `r5` in the code that it generates. This is the default.

`-mbig-switch`: Specifying this option tells GCC to generate code suitable for big switch tables. You should only use this option if the assembler or linker complains about out-of-range branches within a switch table.

`-mdisable-callt`: Specifying this option prevents GCC from using the callt instruction in the code that it generates for the V850e and V850e1 processors.

`-mep`: Specifying this option tells GCC to optimize basic blocks that use the same index pointer four or more times to copy pointers into the `ep` register, using the shorter sld and sst instructions instead. This option is on by default if you specify any of the optimization options.

`-mlong-calls`: Specifying this option tells GCC to treat all calls as being far. If calls are assumed to be far, GCC will always load the function's address into a register and make an indirect call through the pointer.

`-mno-app-regs`: Specifying this option tells GCC to treat `r2` and `r5` as fixed registers in the code that it generates.

`-mno-disable-callt`: Specifying this option enables GCC to use the callt instruction in the code that it generates for the V850e and V850e1 processors as needed. This is the default.

`-mno-ep`: Specifying this option tells GCC not to optimize basic blocks that use the same index pointer four or more times to copy pointers into the `ep` register.

`-mno-long-calls`: Specifying this option tells GCC to treat all calls as being near, requiring no special handling.

`-mno-prolog-function`: Specifying this option tells GCC not to use external functions to save and restore registers at the prologue and epilogue of a function.

`-mprolog-function`: Specifying this option tells GCC to use external functions to save and restore registers at the prologue and epilogue of a function. These external functions are slower but use less code space if more than one function saves the same number of registers. This option is enabled by default if you specify any of the optimization options.

`-msda=`*n*: Specifying this option tells GCC to put static or global variables whose size is *n* bytes or less into the small data area that register `gp` points to. The small data area can hold up to 64K.

`-mspace`: Specifying this option tells GCC to try to make the code as small as possible by activating the `-mep` and `-mprolog-function` options.

`-mtda=`*n*: Specifying this option tells GCC to put static or global variables whose size is *n* bytes or less into the tiny data area that register `ep` points to. The tiny data area can hold up to 256 bytes in total (128 bytes for byte references).

`-mv850`: Specifying this option tells GCC that the target processor is the V850 and defines the preprocessor symbols `__v850` and `__v850__`.

`-mv850e`: Specifying this option tells GCC that the target processor is the V850e, and defines the preprocessor symbol `__v850e__` as well as the standard symbols `__v850` and `__v850__`.

`-mv850e1`: Specifying this option tells GCC that the target processor is the V850E1, and defines the preprocessor symbols `__v850e1__` and `__v850e__`, as well as the standard symbols `__v850` and `__v850__`.

■ **Note** If none of the `-mv850`, `-mv850e`, or `-mv850e1` options are specified, GCC will select a default processor depending on how it was built and define the appropriate symbols. The standard symbols `__v850` and `__v850__` are always defined.

`-mzda=`*n*: Specifying this option tells GCC to put static or global variables whose size is *n* bytes or less into the first 32K of memory.

VAX Options

VAX was Digital Equipment Corporation's (DEC) workstations and minicomputers for years, and shipped with a quaint operating system called VMS.

GCC options available when compiling code for VAX systems are the following:

`-mg`: Specifying this option tells GCC to generate code for g-format floating-point numbers instead of d-format floating-point numbers.

`-mgnu`: Specifying this option tells GCC to generate all possible jump instructions. This option assumes that you will assemble your code using the GNU assembler.

`-munix`: Specifying this option tells GCC not to generate certain jump instructions (aobleq and so on) that the Unix assembler for the VAX cannot handle across long ranges.

Xstormy16 Options

Sanyo's Xstormy16 processor is designed for memory-constrained applications, and is often used in home appliances and audio systems.

The sole GCC option available when compiling code for Xstormy16 processors is the following:

`-msim`: Specifying this option tells GCC to choose startup files and linker scripts that are suitable for an Xstormy16 simulator.

Xtensa Options

The Xtensa architecture is designed to support many different configurations—there is no such thing as a generic Xtensa processor. Each instance of the Xtensa processor architecture is uniquely tuned by the system designer to ideally fit the application targeted by a specific system-on-a-chip (SoC) implementation, by using the Xtensa Processor Generator or by selecting from a broad selection of predefined standard RISC microprocessor features.

GCC's default options can be set to match a particular Xtensa configuration by copying a configuration file into the GCC sources when building GCC.

GCC options available to override the default values specified in the configuration file when compiling code for Xtensa processors are the following:

`-mbig-endian`: Specifying this option tells GCC to generate code using big endian byte ordering. This option is only supported in GCC 3.x versions of the Xtensa compilers.

`-mbooleans`: Specifying this option tells GCC to enable support for the Boolean register file used by Xtensa coprocessors. This is not typically useful by itself but may be required for other options that make use of the Boolean registers (such as the floating-point options). This option is only supported in GCC 3.x versions of the Xtensa compilers.

`-mconst16`: Specifying this option enables GCC to generate code that uses the const16 instruction to load values as needed. The const16 instruction is not a standard Tensilica instruction but replaces the standard l32r instruction when this option is specified. This option is the default if the l32r instruction is not available.

`-mdensity`: Specifying this option tells GCC to enable the use of the optional Xtensa code density instructions. This option is only supported in GCC 3.x versions of the Xtensa compilers.

`-mfused-madd`: Specifying this option tells GCC to enable the use of fused multiply/add and multiply/subtract instructions in the floating-point option. This has no effect if the floating-point option is not also enabled. This option should not be used when strict IEEE 754–compliant results are required, since the fused multiply/add and multiply/subtract instructions do not round the intermediate result, and therefore produce results with more precision than is specified by the IEEE standard.

`-mhard-float`: Specifying this option tells GCC to enable the use of the floating-point option. GCC generates floating-point instructions for 32-bit float operations. 64-bit `double` operations are always emulated with calls to library functions. This option is only supported in GCC 3.x versions of the Xtensa compilers.

`-mlittle-endian`: Specifying this option tells GCC to generate code using little endian byte ordering. This option is only supported in GCC 3.x versions of the Xtensa compilers.

`-mlongcalls`: Specifying this option tells GCC to instruct the assembler to translate direct calls to indirect calls unless it can determine that the target of a direct call is in the range allowed by the call instruction. This translation typically occurs for calls to functions in other source files. This option should be used in programs where the call target can potentially be out of range. This option is implemented in the assembler, not the compiler, so the assembly code generated by GCC will still show direct call instructions; you will have to examine disassembled object code in order to see the actual instructions. This option also causes the assembler to use an indirect call for every cross-file call, not just those that really will be out of range.

`-mmac16`: Specifying this option tells GCC to enable the use of the Xtensa MAC16 option. GCC will generate MAC16 instructions from standard C code, with the limitation that it will use neither the `MR` register file nor any instruction that operates on the `MR` registers. This option is only supported in GCC 3.x versions of the Xtensa compilers.

`-mminmax`: Specifying this option tells GCC to enable the use of the optional minimum and maximum value instructions. This option is only supported in GCC 3.x versions of the Xtensa compilers.

`-mmul16`: Specifying this option tells GCC to enable the use of the 16-bit integer multiplier option. GCC will generate 16-bit multiply instructions for multiplications of 16 bits or smaller in standard C code. This option is only supported in GCC 3.x versions of the Xtensa compilers.

`-mmul32`: Specifying this option tells GCC to enable the use of the 32-bit integer multiplier option. GCC will generate 32-bit multiply instructions for multiplications of 32 bits or smaller in standard C code. This option is only supported in GCC 3.x versions of the Xtensa compilers.

`-mno-booleans`: Specifying this option tells GCC to disable support for the Boolean register file used by Xtensa coprocessors. This option is only supported in GCC 3.x versions of the Xtensa compilers.

`-mno-const16`: Specifying this option prevents GCC from generating code that uses the const16 instruction to load values, using the standard l32r instruction instead.

`-mno-density`: Specifying this option tells GCC to disable the use of the optional Xtensa code density instructions. This option is only supported in GCC 3.x versions of the Xtensa compilers.

`-mno-longcalls`: Specifying this option tells GCC not to translate direct calls to indirect calls. This is the default. See the description of the `-mlongcalls` option for implementation details.

`-mno-mac16`: Specifying this option tells GCC to disable the use of the Xtensa MAC16 option. GCC will translate 16-bit multiply/accumulate operations to a combination of core instructions and library calls, depending on whether any other multiplier options are enabled. This option is only supported in GCC 3.x versions of the Xtensa compilers.

`-mno-fused-madd`: Specifying this option tells GCC to disable the use of fused multiply/add and multiply/subtract instructions in the floating-point option. Disabling fused multiply/add and multiply/subtract instructions forces the compiler to use separate instructions for the multiply and add/subtract operations. This may be desirable in some cases where strict IEEE 754–compliant results are required, since the fused multiply/add and multiply/subtract instructions do not round the intermediate result, thereby producing results with more precision than is specified by the IEEE standard. Disabling fused multiply/add and multiply/subtract instructions also ensures that the program output is not sensitive to the compiler's ability to combine multiply and add/subtract operations.

`-mno-minmax`: Specifying this option tells GCC to disable the use of the optional minimum and maximum value instructions. This option is only supported in GCC 3.x versions of the Xtensa compilers.

`-mno-mul16`: Specifying this option tells GCC to disable the use of the 16-bit integer multiplier option. GCC will either use 32-bit multiply or MAC16 instructions if they are available, or generate library calls to perform the multiply operations using shifts and adds. This option is only supported in GCC 3.x versions of the Xtensa compilers.

`-mno-mul32`: Specifying this option tells GCC to disable the use of the 32-bit integer multiplier option. GCC will generate library calls to perform the multiply operations using either shifts and adds or 16-bit multiply instructions if they are available. This option is only supported in GCC 3.x versions of the Xtensa compilers.

`-mno-nsa`: Specifying this option tells GCC to disable the use of the optional normalization shift amount (NSA) instructions to implement the built-in `ffs` function. This option is only supported in GCC 3.x versions of the Xtensa compilers.

`-mno-serialize-volatile`: Specifying this option tells GCC not to use MEMW instructions before volatile memory references in order to guarantee sequential consistency, resulting in decreased code size.

`-mno-sext`: Specifying this option tells GCC to disable the use of the optional sign extend (SEXT) instruction. This option is only supported in GCC 3.x versions of the Xtensa compilers.

`-mno-target-align`: Specifying this option tells GCC to not have the assembler automatically align instructions, increasing code density. Specifying this option does not affect the treatment of autoaligned instructions such as LOOP, which the assembler will always align, either by widening density instructions or by inserting NOOP instructions.

`-mno-text-section-literals`: Specifying this option tells GCC to place literals in a separate section in the output file. This allows the literal pool to be placed in a data RAM/ROM, and also allows the linker to combine literal pools from separate object files to remove redundant literals and improve code size. This is the default.

`-mnsa`: Specifying this option tells GCC to enable the use of the optional NSA instructions to implement the built-in `ffs` function. This option is only supported in GCC 3.x versions of the Xtensa compilers.

`-mserialize-volatile`: Specifying this option tells GCC to insert MEMW instructions before volatile memory references in order to guarantee sequential consistency. This option is only supported in GCC 3.x versions of the Xtensa compilers, where this option is the default.

`-msext`: Specifying this option tells GCC to enable the use of the optional SEXT instruction. This option is only supported in GCC 3.x versions of the Xtensa compilers.

`-msoft-float`: Specifying this option tells GCC to disable use of the floating-point option. GCC generates library calls to emulate 32-bit floating-point operations using integer instructions. 64-bit `double` operations are always emulated with calls to library functions. This option is only supported in GCC 3.x versions of the Xtensa compilers.

`-mtarget-align`: Specifying this option tells GCC to instruct the assembler to automatically align instructions in order to reduce branch penalties at the expense of some code density. The assembler attempts to widen density instructions to align branch targets and the instruction's following call instructions. If there are not enough preceding safe density instructions to align a target, no widening will be performed. This is the default.

`-mtext-section-literals`: Specifying this option tells GCC to intersperse literals in the text section in order to keep them as close as possible to their references. This may be necessary for large assembly files.

APPENDIX C

Using GCC's Online Help

For better or for worse, the FSF uses a nonstandard online help format, GNU Info, to document its software. GNU Info is one of those utilities people love to hate. Not only is GNU Info a more powerful and flexible system for documenting software than the ubiquitous Unix manual page, it is also dramatically different, which has kept it from becoming popular. Indeed, if you want to start a flame war on your favorite mailing list, don your asbestos underwear and post a message stating that Info is better than man. Seriously, because GNU Info is not very popular, most people do not now how to use it. Unfortunately, most of GCC's excellent documentation is maintained in GNU Info format; so to get the most out of GCC's own documentation, you have to know how to use GNU Info—this chapter teaches you how to do so. I think you will find that it is not nearly as difficult to use as it seems.

What Is GNU Info?

What is GNU Info? The simple answer is that GNU Info is a hypertext online help system intended by the FSF to replace the traditional (and, I dare say, the standard) Unix manual (or man) page. GNU Info enables online help systems to use tables of contents, cross-references, and indexes without requiring the overhead of an HTML browser. Info was originally designed long before the explosion of the Internet and before the World Wide Web became ubiquitous. Because of its hypertext abilities, GNU Info makes it easy to jump from one reference page to another that contains related information and then to return to the original page. This hyperlinking is a clear advantage over the traditional Unix manual page.

Unfortunately, the Info user interface, which bears a close resemblance to the Emacs user interface, is substantially different from and far more complicated than the interface used by traditional Unix manual page viewers, which are usually some variant of the standard text viewing programs, more or less. GNU's stubborn refusal to make traditional manual pages available for most of its software (with some notable exceptions, of course—the Bash manual pages come to mind) and Info's different interface has prevented it from becoming an accepted, popular replacement for the more limited but easier to use Unix manual page. So this short chapter will show you how to use GNU Info while staying out of the Info vs. man dispute.

Note I confess that I started writing this chapter from the perspective of a confirmed manual page advocate and committed Info hater. But my opinion of GNU Info has changed. Although I still wish that the GNU project would make both manual and Info pages available instead of forcing users to use Info, I have found that Info gives users greater capabilities for searching and viewing related information than the current man implementations. I still use the `man` command for quick reference information, but I no longer swear a blue streak if I have to use GNU Info.

GNU Info is more than just a hypertext online help system. It is really a complete document production system. The `info` command is a program that knows how to view and navigate Info files. Info files, in turn, are Texinfo (pronounced "teck-info") format text files that have been processed by the GNU `makeinfo` command in order to create Info files that the `info` command can read. Texinfo source files are plain ASCII text files containing so-called *@-commands*, which are characters and words prefixed with @, and a few other characters with special meanings. The @-commands and other special characters provide instructions to formatting and typesetting programs about how to format and typeset the text.

One of the attractions of Texinfo is that a single Texinfo source file can be used to produce Info files for use with GNU Info, typeset files ready for printing, and HTML output that you can browse with any Web browser. For the curious, Listing C-1 shows a small sample of Texinfo source code taken from the invoke.texi file. This short example of Texinfo source code describes the GCC keywords you can use when invoking the C compiler with the `-ansi` option. Listing C-2 shows the same Texinfo source code after it has been rendered into an Info file and displayed using the Info browser, `info`.

Listing C-1. *Sample Texinfo Source Code*

```
The alternate keywords @code{__asm__}, @code{__extension__},
@code{__inline__} and @code{__typeof__} continue to work despite
@option{-ansi}.  You would not want to use them in an ISO C program, of
course, but it is useful to put them in header files that might be included
in compilations done with @option{-ansi}.  Alternate predefined macros
such as @code{__unix__} and @code{__vax__} are also available, with or
without @option{-ansi}.

The @option{-ansi} option does not cause non-ISO programs to be
rejected gratuitously.  For that, @option{-pedantic} is required in
addition to @option{-ansi}.  @xref{Warning Options}.
```

Listing C-2. *Texinfo Source Code After Rendering*

```
The alternate keywords `__asm__', `__extension__', `__inline__'
and `__typeof__' continue to work despite `-ansi'.  You would not
want to use them in an ISO C program, of course, but it is useful
to put them in header files that might be included in compilations
done with `-ansi'.  Alternate predefined macros such as `__unix__'
and `__vax__' are also available, with or without `-ansi'.

The `-ansi' option does not cause non-ISO programs to be rejected
gratuitously.  For that, `-pedantic' is required in addition to
`-ansi'.  *Note Warning Options::.
```

I strongly recommend that you learn how to use the Info interface because a great deal of valuable information, especially about GCC, is only accessible in Info files. The next section, "Getting Started, or Instructions for the Impatient," should be sufficient to get you started. Subsequent sections go into detail and offer variations and extensions of the techniques discussed in the next section.

Getting Started, or Instructions for the Impatient

To start using GNU Info, type **info *program***, replacing *program* with the name of the program that interests you. This being a book about GCC, you might try

```
$ info gcc
```

To exit Info, type **q** to return to the command prompt.

If you are staring at an Info screen right now and just want to get started, Table C-1 lists the basic Info navigation commands you'll need to find your way around the Info help system.

Table C-1. *Basic Info Navigation Commands*

Key or Key Combination	Description
Ctrl-n	Moves the cursor down to the next line
Down	Moves the cursor down to the next line
Ctrl-p	Moves the cursor up to the previous line
Up	Moves the cursor up to the previous line
Ctrl-b	Moves the cursor one character to the left
Left	Moves the cursor one character to the left
Ctrl-f	Moves the cursor one character to the right
Right	Moves the cursor one character to the right
Ctrl-a	Moves the cursor to the beginning of the current line
Ctrl-e	Moves the cursor to the end of the current line
Spacebar	Scrolls forward (down) one screen or advances to the next screen if you are at the bottom of the current one
Ctrl-v	Scrolls forward (down) one screen
Page Down	Scrolls forward (down) one screen
Meta-v	Scrolls backward (up) one screen
Page Up	Scrolls backward (up) one screen
b	Moves the cursor to the beginning of the current node
e	Moves the cursor to the end of the current node
l	Moves to the most recently visited node
n	Moves to the next node
p	Moves to the previous node
u	Moves up a node
s	Performs a case-insensitive search for a string
S	Performs a case-sensitive search for a string
Ctrl-g	Cancels iterative operations such as an incremental search
Ctrl-x n	Repeats the last search
Ctrl-x N	Repeats the last search in the opposite direction

The notation Ctrl-x means to press and hold the Control key while pressing the x key. Meta-x means to press and hold the Meta key (also known as the Alt key) while pressing the x key. Finally, Ctrl-x n means to press and hold the Control key while pressing the x key, release them, and then press the n key.

■ **Tip** If the Alt key combinations described in this section do not work on your system, you can press the Esc key instead. So, for example, Meta-v (or Alt-v) can also be invoked as Esc-v. Notice that if you use the Esc key, you do not have to press and hold it as you do when using the Alt key.

The Emacs users in the audience will recognize the keystrokes in Table C-1 as Emacs key bindings. If you need more help than the keystroke reference in Table C-1 offers you, read the next few sections. If you get totally lost or confused, just type **q** to exit `info` and return to the command prompt.

Getting Help

Predictably, GNU Info has a help screen, accessible by pressing Ctrl-h. Use the Info commands discussed in this chapter to navigate the help screen. When you want to close the help window, press the l key—the help window is just a specially handled node pointer, so the l key returns you to the node you were originally viewing. If you are so inclined, you can go through the Info tutorial by pressing Ctrl-h within a screen. Remember the magic rescue sequence: if all else fails and you find yourself totally disoriented, press the q key, possibly several times, to exit Info.

The Beginner's Guide to Using GNU Info

This section explains the concepts and commands that cover 80 percent of GNU Info usage. By the time you have completed this section of the chapter, you will know more about GNU Info and how to use it more effectively than almost everyone else who uses GNU software. The other 20 percent of GNU Info commands, covered in the section titled "Stupid Info Tricks," explore advanced features and techniques whose use arguably constitute Info mastery. Okay, perhaps mastery is a bit over the top, but the "Stupid Info Tricks" section at the end of the chapter does describe sophisticated methods you will not use very often.

Anatomy of a GNU Info Screen

This section describes the standard components of an Info screen. You'll need to grasp this material because most of the discussion that follows assumes you know what a typical Info file looks like when displayed using the `info` command. Figure C-1 shows a typical Info screen (as luck would have it, it is the top-level node of the GCC Info file), with the various elements described in the text identified by callouts.

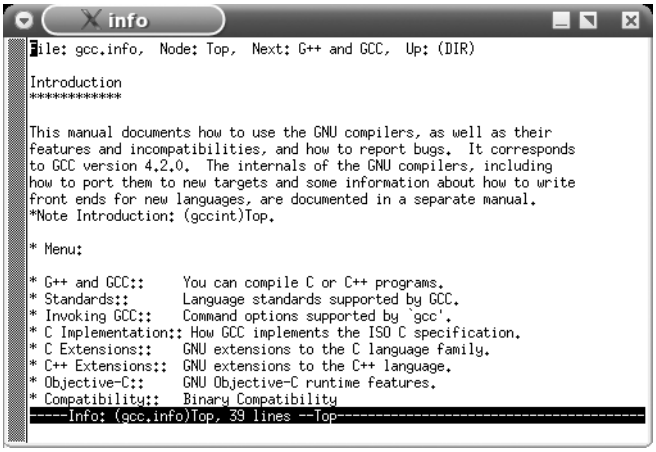

info
File: gcc.info, Node: Top, Next: G++ and GCC, Up: (DIR)
Introduction
This manual documents how to use the GNU compilers, as well as their
features and incompatibilities, and how to report bugs. It corresponds
to GCC version 4.2.0. The internals of the GNU compilers, including
how to port them to new targets and some information about how to write
front ends for new languages, are documented in a separate manual.
*Note Introduction: (gccint)Top.
* Menu:
* G++ and GCC:: You can compile C or C++ programs.
* Standards:: Language standards supported by GCC.
* Invoking GCC:: Command options supported by `gcc'.
* C Implementation:: How GCC implements the ISO C specification.
* C Extensions:: GNU extensions to the C language family.
* C++ Extensions:: GNU extensions to the C++ language.
* Objective-C:: GNU Objective-C runtime features.
* Compatibility:: Binary Compatibility
-----Info: (gcc.info)Top, 39 lines --Top-----

Figure C-1. *A typical GNU Info screen*

In general, GNU Info uses the term *window* to describe the screen area that displays Info text. Each such window has a mode line at the bottom that describes the node being displayed. The information displayed in the mode line includes the following information, reading left to right:

- The name of the file
- The name of the node
- The number of screen lines necessary to display the node
- The amount of text above the current screen, expressed as a percentage

So in Figure C-1, the name of the file is gcc.info.gz; the name of the node is Top; it takes 40 lines to display the entire screen; and the screen currently is displaying the top of the file. The text "zz" at the beginning of the mode line indicates that the file was read from a compressed disk file. The text beginning with "Subfile" means that this node has been split into multiple files and that, in this case, you are viewing the subfile gcc.info-1.gz. This text does not appear if the Info node has been spread across several files.

The documentation for GNU Info (available, oddly enough, as an Info file—type **info info** to view it) refers to the screen real estate occupied by actual Info text as the view area. However, I prefer to use the term *window* because the `info` command is capable of displaying multiple windows inside one screen and I do not want to have to refer to "the view area in the foo window" if "the foo window" is a clear enough reference. When the `info` command displays multiple windows in the same screen, one window is separated from another by its mode line, and the mode line always marks the bottom of a given window.

An echo area appears on the last or bottom line of each Info screen. The echo area, also called an *echo line*, displays status information and error messages, and serves as a buffer for typing the text used for word searches or for other input you might have to provide. When you are at the top or at the beginning of a node, a line of text across the top of the screen, which I call a *node pointer*, identifies the current file and node, and the next, previous, and parent nodes of the current node, when applicable. Node pointers and their usage are described in the next section, "Moving Around in GNU Info."

The topics listed in the center of the screen in Figure C-1 are known as *menu references*. If you move the cursor to any character between * and : : and press Enter, a process called *selecting a node*, you will jump to the Info file corresponding to that topic (commonly called a *node*). For example, Figure C-2 shows the screen that appears if you select the topic "C Extensions" and then, on that screen, select the topic "Labels as Values."

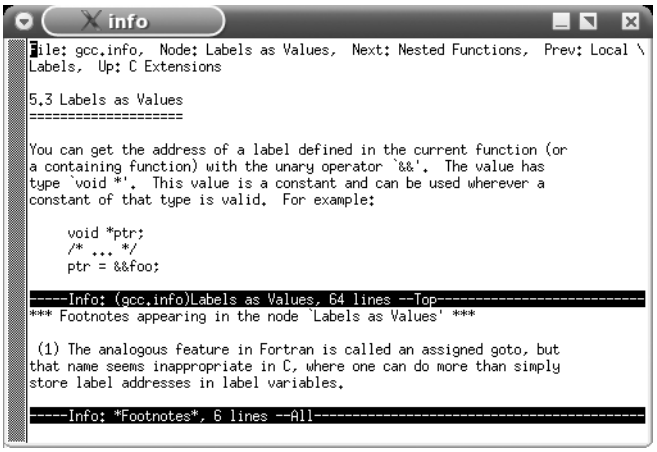

info
File: gcc.info, Node: Labels as Values, Next: Nested Functions, Prev: Local \
Labels, Up: C Extensions
5.3 Labels as Values
You can get the address of a label defined in the current function (or
a containing function) with the unary operator `&&'. The value has
type `void *'. This value is a constant and can be used wherever a
constant of that type is valid. For example:
void *ptr;
/* ... */
ptr = &&foo;
-----Info: (gcc.info)Labels as Values, 64 lines --Top-----
*** Footnotes appearing in the node `Labels as Values' ***
(1) The analogous feature in Fortran is called an assigned goto, but
that name seems inappropriate in C, where one can do more than simply
store label addresses in label variables.
-----Info: *Footnotes*, 6 lines --All-----

Figure C-2. *Following a GNU Info menu reference*

Notice that the screen shown in Figure C-2 has two windows, each separated from the other by a mode line as described earlier in this appendix. The upper window contains Info text, and the bottom window contains footnotes that appear in this node. If you press the spacebar, the text in the upper window scrolls forward one page or screen, but the footnote window remains static. You will learn later in this chapter how to navigate between windows. You might also notice that the mode line for the upper window indicates that you have jumped to the subfile gcc.info-11.gz.

Moving Around in GNU Info

You can get a lot done in GNU Info files just by pressing the spacebar (or Ctrl-v) to scroll through each screen in order. If you want to go backward, use Meta-v. If your keyboard has them, you can probably use the Next and Previous keys (also known as Page Down and Page Up on standard PC keyboards), instead of Ctrl-v and Meta-v. PC users will find Next and Previous more convenient and familiar. In all cases, though, the canonical sequences (Ctrl-v and Meta-v) should work. Similarly, the cursor keys allow you to scroll up and down through the screen you are currently looking at, one line of text at a time (in this case, the canonical keys are Ctrl-n for up and Ctrl-p for down), or to move left and right on the current line of text using Ctrl-b and Ctrl-f (left and right). Most PC users might find the cursor movement keys the most familiar; but the Control key sequences should always work regardless of the keyboard.

The node movement keys b, e, l, n, p, and u are easiest to understand if you are looking at the top of an Info page. For example, if you type **info gcc invoking** at a command prompt, the first few lines should resemble Listing C-3, the first page of the Invoking GCC node.

Listing C-3. *The First Page of the GCC Info File*

```
File: gcc.info,  Node: Invoking GCC,  Next: Installation,  Prev: G++ and GCC,  \
Up: Top

GCC Command Options
*******************
```

The line of text across the top shows the name of the file containing the node you are currently viewing (gcc.info), the name of the node itself (Node: Invoking GCC), the next node (Next: Installation), the previous node (Prev: G++ and GCC), and the top of the node tree (Up: Top). A node (more precisely, a node pointer) is an organizational unit in Info files that corresponds to chapters, sections, subsections, footnotes, or index entries in a printed book. It also makes it easier to navigate in an Info file. As Table C-1 shows, n jumps to the next node, p jumps to the previous node, and u jumps up a node or to the parent node. So, from the screen illustrated in Listing C-3, typing n jumps to the Installation node, p jumps to the node title G++ and GCC, and u takes you to the Top node, which, in this case, is the main menu or table of contents for the GCC Info file. To put it another way, the Top node is the parent of the current node, which may or may not always be the main menu. However, it usually only takes a few presses of the u key to get to the top of the tree.

Within a node, typing **b** places the cursor at the top of the current node and typing **e** places you at the end of the current node. I think the l (last node) key performs one of the most useful jumps, because it takes you back to the most recently visited node (the last node). The l key is comparable to the Back button in a Web browser. Figure C-3 illustrates how a complete Info topic might be organized.

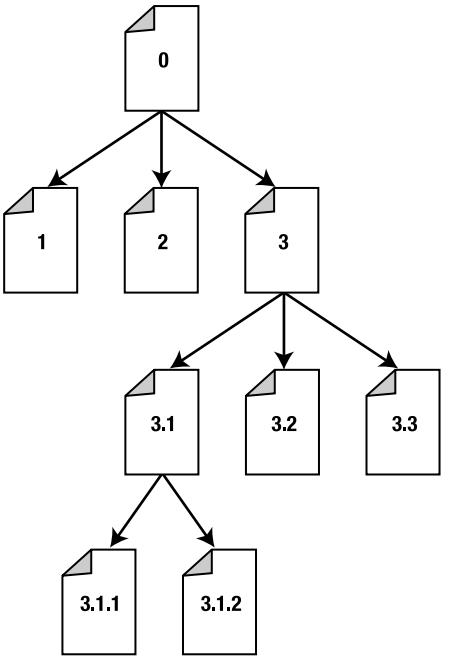


Figure C-3. *The node structure of a GNU Info file*

There are three Top nodes in Figure C-3. The Top node for 3.1.1 and 3.1.2 is 3.1. The Top node for 3.1, 3.2, and 3.3 is 3, and 3's Top node, like 1 and 2, is 0. So, if you were viewing 3.1.2, you would type **u** three times to return to the top or root node. If Figure C-3 were arranged as a table of contents, it would resemble the following:

```
Chapter 1
Chapter 2
Chapter 3
    Section 3.1
        Subsection 3.1.1
        Subsection 3.1.2
    Section 3.2
    Section 3.3
```

Finally, if you walked through the nodes using the n key or the spacebar, the progression would be 0 -> 1 -> 2 -> 3 -> 3.1 -> 3.1.1 -> 3.1.2 -> 3.2 -> 3.3.

I have made such an effort to explain and illustrate the node structure of a GNU Info file and how to navigate through it because most users unfamiliar with Info complain about getting lost in it and not knowing where they are, how to get back to where they were, and where they want to be. It can be boiled down to two simple rules:

1. Learn the mnemonics of the b, e, n, p, and u keys: (b)eginning of node, (e)nd of node, (n)ext node, (p)revious node, and (u)p to the Top node.
2. Keep your eye on the node pointers at the top of a node so you will know where the n, p, and u keys will take you.

Performing Searches in GNU Info

One of the most useful features of GNU Info is its ability to perform fast searches and a large variety of searches. The stand-alone `info` command can search forward and backward searches and searches that are case sensitive or insensitive. You can search incrementally or by complete string, and you can also search an Info file's indices and then jump to the node containing the matching index entry. Although man page viewers have similar functionality, I have found GNU Info's search features to be faster and more capable. As always, your mileage may vary. Before jumping into the details of searches, Table C-2 lists the search functions you need to know.

Table C-2. *GNU Info Search Commands*

Key or Key Combination	Description
?	Searches backward for a string typed in the echo area, ignoring case.
,	Jumps to the next node containing a match for a previously completed index search.
i	Searches the file index for a string typed in the echo area and jumps to the node containing a match.
s	Searches for a string typed in the echo area, ignoring case.
S	Searches for a string typed in the echo area, paying attention to case.
Ctrl-g	Aborts the current search.
Ctrl-r	Searches backward for a string as you type it.

Table C-2. *GNU Info Search Commands*

Key or Key Combination	Description
Ctrl-s	Searches forward for a string as you type it.
Ctrl-x n	Repeats the previous search, retaining the previous search's case sensitivity.
Ctrl-x N	Repeats the previous search in the reverse direction, retaining the previous search's case sensitivity.

While performing a forward search with Ctrl-s, you can press Ctrl-s to search forward for the next occurrence, or Ctrl-r to search backward for the previous occurrence. Conversely, when running a backward search using Ctrl-r, pressing Ctrl-r repeats the backward search for the next (previous) occurrence, and Ctrl-s executes the same search in the forward direction.

Another way to repeat the previous search is to type **s** (or **S**) and press Enter to accept the default string, which is always the last search string embedded in square brackets ([]). Pressing Enter without typing a new search string uses the default value. When repeating previous searches using Ctrl-x n or Ctrl-x N, the previous search's case sensitivity option is applied. The search features Info provides do show one small bit of perversity: the s search is case insensitive unless the string entered contains an uppercase letter, in which case the search automatically becomes case sensitive.

■ **Tip** In almost every mode, pressing Ctrl-g, sometimes several times, will cancel most operations and restore Info's interface to something sane.

Following Cross-References

Another one of GNU Info's most useful features, and the source of much gnashing of teeth for the uninitiated, is its support for cross-references, referred to as *xrefs* in the GNU Info documentation. xrefs are pointers or hotlinks to other Info nodes that contain related information (and that may exist in a separate file). It is these xrefs that give Info its hypertext capability. It is also these xrefs that cause users to get lost and start hating Info. And it is when using these xrefs that the l key becomes most useful—getting you back to where you started. A canonical xref has the following form:

```
* label: target. comment
```

An xref begins with *, followed by *label*, which is a name that you can use to refer to the xref; this is followed by a colon, which separates the label from the target; the target itself; a terminating period to end the target name; and an optional *comment*. *target* is the full name of the node to which the xref refers. The optional comment is often used to describe the contents of the referenced node. Consider the following xref taken from the GCC Info file's index:

```
* allocate_stack instruction pattern:    Standard Names.
```

In this example, *label* is the text "allocate_stack instruction pattern" and *target* is the node named *Standard Names*. The period (.) is not part of the target, but merely indicates the end of the target name. To select an xref, position the cursor anywhere between the * and the end of the target name and then press Enter.

You will often see xrefs that use the following form:

```
* G++ and GCC::     You can compile C or C++ programs.
```

In this case, the target name has been omitted and a second colon used in its place. This means that the label and the target name have the same names. That is, the previous xref is equivalent to

```
* G++ and GCC:G++ and GCC.     You can compile C or C++ programs.
```

This shorthand notation is especially common in node menus, such as the one shown in Figure C-1. In fact, this shorthand notation is so commonly used in node menus that it is called a *menu reference* in order to distinguish from the other standard type of xref, a *note reference*. Unlike menu references, which appear in, you guessed it, node menus, note references appear in the body text of a node, inline with the rest of the text, much like the hyperlinks in the body text of a standard HTML Web page. Note references are very similar to menu references, except that they begin with *Note, rather than just a bare *. Listing C-4, for example, extracted from the GCC Info file, contains a note reference.

Listing C-4. *A Typical Note Reference*

```
element of the union should be used.  For example,

     union foo { int i; double d; };

     union foo f = { d: 4 };

will convert 4 to a 'double' to store it in the union using the second
element.  By contrast, casting 4 to type 'union foo' would store it
into the union as the integer 'i', since it is an integer.  (*Note Cast
to Union::.)
```

The note reference appears at the end of the listing and refers to the node named *Cast to Union*, which explains the semantics of casting a type to a C union type. After selecting either type of xref, you can use the l key to return to the node you were reading before you jumped, or you can follow another reference. No matter how far afield you stray, if you press l enough times, you will eventually traverse the history list the `info` command maintains until you return to the node from which you started.

Printing GNU Info Nodes

GNU Info's documented method for printing an individual node is the command `M-x print-node`, which pipes the contents of the node to the `lpr` command. If the environment variable `INFO_PRINT_COMMAND` is set, the node's contents will be piped through the command defined in this variable. However, if you want to print a complete Info file rather than a single node, you might want to use the following command:

```
$ info --subnodes info_file -o - | lpr
```

This command prints the entire Info file named info_file using the default system printer. The `--subnodes` option tells the `info` command to recurse through each node of info_file's menus and display the contents of each node as it recurses through the node tree. The `-o` option tells the `info` command to save the output in a file and must be followed by a filename. A filename of - specifies standard output. In the command shown, output is piped into `lpr`, which, if your printer is properly configured, will print the entire manual. Thus, to print the entire Info file for the `info` command, the command would be

```
$ info --subnodes info -o - | lpr
```

■ **Caution** You may not want to actually print this, since the entire Info manual is 24 pages long.

Invoking GNU Info

The complete syntax for executing GNU Info from the command line is

`info [`*`options`*`...] [`*`info_file`*`...]`

info_file identifies the Info node(s) you want to view, separated by whitespace. For example, `info gcc` opens the GCC Info file. If you want to visit a particular menu item or subnode available from the menu (table of contents) in the GCC Info file, specify it after the top-level node. So if you want to go straight to the Installation node of the GCC Info file, use the command `info gcc installation`. Additional arguments will take you deeper into the node tree. Thus, `info gcc installation cross-compiler` will take you to the Cross-Compiler subnode of the Installation subnode of the gcc node. The case of the specified nodes and subnodes is ignored. Table C-3 lists the values for the options arguments that I find most useful—the complete list is available in the Invoking Info node of the `info` command's Info file (info info invoking).

Table C-3. *Common GNU Info Command–Line Options*

Option	Description
`--apropos=`*`word`*	Starts `info` looking for *`word`*.
`-d` *`dir`*	Adds the directory *`dir`* to the `info` program's search path.
`--directory=`*`dir`*	Adds the directory *`dir`* to the `info` program's search path.
`--dribble=`*`file`*	Logs keystrokes to *`file`* (see `--restore`).
`-f` *`file`*	Opens the Info file specified by *`file`*, bypassing the special directory file and the default search path.
`--file=`*`file`*	Opens the Info file specified by *`file`*, bypassing the special directory file and the default search path.
`-h`	Displays a short usage message.
`--help`	Displays a short usage message.
`--index-search=`*`string`*	Searches the specified Info file's index node for *`string`* and jumps to the first matching node.
`--location`	Displays the full path name of the Info file that would be used, and exits.
`-n` *`node`*	Goes directly to the specified node in the Info file, if it exists.
`--node=`*`node`*	Goes directly to the specified node in the Info file, if it exists.
`-O`	Jumps directly to an Info node that identifies how to invoke a program, if such a node exists.
`-o` *`file`*	Redirects Info output from the display to the file specified by *file*.
`--output` *`file`*	Redirects Info output from the display to the file specified by *file*.

Table C-3. *Common GNU Info Command–Line Options (Continued)*

Option	Description
`-R`	Leaves ANSI formatting on the page
`--raw-escapes`	Leaves ANSI formatting on the page.
`--restore=`*`file`*	Reads keystrokes from *`file`* (complements `--dribble`).
`--show-options`	Same as `-O` and `--usage`.
`--subnodes`	Recursively displays the child nodes of the top-level Info file (must be used with the `-o` option to specify an output file).
`--usage`	Jumps directly to an Info node that identifies how to invoke a program, if such a node exists.
`--version`	Displays `info` version information and exits.
`--vi-keys`	Starts `info` using key bindings that work somewhat like `vi`.
`-w`	Displays the full pathname of the Info file that would be used, and exits.
`--where`	Displays the full pathname of the Info file that would be used, and exits.

If `-f` *`file`* specifies an absolute pathname, such as `-f ./gcc.info`, Info will only look at the specified file or path, ignoring the built-in search path. If a search specified by `--index-search` fails, Info displays an error message. As explained earlier, a bare `-` specified with `-o` is interpreted as standard output, which allows you to use Info output in pipelines and with the `--subnodes` option.

Stupid Info Tricks

Despite the title, this section offers a few clever tips and hints for using GNU Info. Learning to use the information in this section might turn you into a certified GNU Info power user, allowing you to dazzle your friends and colleagues with your mastery of GNU Info.

One of my favorite tricks uses the `--subnodes` option. As you can probably imagine, I spent *a lot* of time studying GCC's Info pages. When I got tired of looking at onscreen text, I used the following shell pipeline, or a variation of it, to print out all or part of the GCC manual.

```
$ info --subnodes gcc -o - | enscript -G -H -o - | psnup -2 | lpr
```

The first part of the command you have already seen. Instead of piping the output straight to the printer, I ran it through enscript to convert the text to PostScript, piped enscript's output to psnup to create two-up pages (to reduce paper consumption), and then printed the result. Enscript's `-G` option adds a flashy header to the top of each (virtual) page; `-H` adds highlight bars to the printed output, emulating what old computer hacks know as *greenbar* (although, in this case it is probably more accurate to call it *graybar*).

Using Command Multipliers

GNU Info, like GNU Emacs, enables you to prefix most commands with a numeric multiplier, causing the commands to execute multiple times. The command that invokes the multiplier is Ctrl-u [*n*], where *n* is the number of times you want the following command to execute. If you omit *n*, the default

value of the multiplier is 4. So, for example, to scroll down four screens, you can press Ctrl-u 4 Ctrl-v, or Ctrl-u Ctrl-v. To scroll backward two screens, you can try Ctrl-u 2 Meta-v, or Ctrl-u -2 Ctrl-v. Yes, that is -2 in the second command—when you use a negative multiplier value, most operations, such as moving or scrolling, work in the opposite or reverse direction. Thus, for example, using a negative multiplier with Ctrl-b causes the cursor to move forward rather than backward.

Working with Multiple Windows

I mentioned earlier in the chapter that GNU Info can display multiple windows in a single screen, with mode lines serving as the visual separators between windows. Info also provides methods for moving from one window to another, changing the size of windows, and for creating and deleting windows. The unspoken question, of course, is "Why would one want to use multiple windows?" The most common reason is to keep from having to jump back and forth between nodes. Instead, you can open an additional window (strictly speaking, you just split the existing window into two separate windows), display a different node in the new window, and then switch between them. This approach allows you to look at the information in two nodes more or less simultaneously. Obviously, it should be clear that you can open as many windows as you want, although too many windows open in the same screen will quickly become unwieldy and difficult to read.

Table C-4 shows common GNU Info commands related to opening, closing, moving between, and resizing Info windows.

Table C-4. *Common Info Commands*

Action	Command	Explanation
Close current window	Ctrl-x 0	Closes the current window and moves the cursor to the next available window.
Close other windows	Ctrl-x 1	Closes all windows other than the one in which the cursor is currently located.
Increase size of current window	Ctrl-x ^	Incrementally increases the size of the current window by one line. Can be used with the Ctrl-u command multiplier to increase window size by multiple lines, as in Ctrl-u 3 Ctrl-x ^, which would increase the size of the current window by three lines.
Move to other window	Ctrl-x o	Moves the cursor to the next, or other, window. If you have more than two windows open, pressing Ctrl-x o repeatedly cycles the cursor through all open windows.
Move to previous window	Esc x o	Moves the cursor to the previous window.
Scroll other window	Esc Ctrl-v	Scrolls the previous/other window forward one screen of text.
Split the current window	Ctrl-x 2	Splits the current window, defined as the window containing the cursor, into two equal-size windows, leaving the cursor in the original window. Info commands, such as cursor movement, scrolling, and following xrefs, only apply to the current window.

Index

A

Acovea
- building and installing, 114
- configuration file, code example, 115–116
- deriving optimal sets of optimization switches, 114
- downloading the source code, 114
- Ladd, Scott, 114
- producing an optimal executable automatically, 114
- runacovea application, 114, 116
- sample output, 116–117
- website, 117

AIX operating system, 474

Alpha options
- Alpha 64-bit processor family, 403
- BWX instruction set, 404, 406
- CIX instruction set, 404, 406
- DEC Unix, 405
- double datatype, 406
- dynamic loaders for shared libraries, 404
- floating-point control register (FPCR), 405
- floating-point register (FPR), 405
- floating-point traps, 408
- hardware floating-point instructions, 407
- IEEE-conformant math library routines, 405
- long double datatype, 406
- -malpha-as, 404
- MAX instruction set, 407
- -mbuild-constants, 404
- -mbwx, 404
- -mcix, 404
- -mcpu=CPU-type, 404
- -mexplicit-relocs, 404
- -mfix, 404
- -mfloat-ieee, 405
- -mfloat-vax, 405
- -mfp-regs, 405
- -mfp-rounding-mode=rounding-mode, 405
- -mfp-trap-mode=trap-mode, 405
- -mgas, 405
- -mieee, 405
- -mieee-conformant, 405
- -mieee-with-inexact, 406
- -mlarge-data, 406
- -mlarge-text, 406
- -mlong-double-128, 406
- -mlong-double-64, 406
- -mmax, 406
- -mmemory-latency=time, 406
- -mno-bwx, 406
- -mno-cix, 406
- -mno-explicit-relocs, 407
- -mno-fix, 407
- -mno-fp-regs, 407
- -mno-max, 407
- -mno-soft-float, 407
- -msmall-data, 407
- -msmall-text, 407
- -msoft-float, 407
- -mtls-kernel, 408

-mtls-size=number, 408
-mtrap-precision=trap-precision, 408
-mtune=CPU-type, 408
Not-a-Number, 405
OSF/1 PAL code, 408
plus/minus infinity, 405
rduniq and wruniq calls, 408
rdval and wrval calls, 408
using malloc or mmap to allocate heap data, 406
Alpha/VMS options
-mvms-return-codes, 408
POSIX-style error codes, 408
VMS operating system, 408
alternate C libraries
adding the correct include directories to the include path, 283
defining all required special symbols, 283
dietlibc, 282
gcc and, 283
Glibc and languages other than C, 282
initramfs filesystem, 282
klibc, 282
Newlib, 282
-nostdlib option, 282
reasons for using, 281
reducing application size and runtime resources, 282
size disadvantage of Glibc, 281
supplying a startup routine to the GCC compiler, 283
uClibc, 282
AMD 29K options
AMD 29000 RISC microprocessor, 409
AMD 29027 floating-point unit (FPU), 409
Berkeley RISC design, 409
-m29000, 409
-m29050, 409
-mbw, 409
-mdw, 409
-mimpure-text, 410
-mkernel-registers, 410
-mlarge, 410
-mnbw, 410
-mndw, 410
-mno-impure-text, 410
-mno-multm, 410
-mno-reuse-arg-regs, 410
-mnormal, 410
-mno-soft-float, 410
-mno-stack-check, 410
-mno-storem-bug, 410
-mreuse-arg-regs, 410
-msmall, 410
-msoft-float, 411
-mstack-check, 411
-mstorem-bug, 411
-muser-registers, 411
small memory model, 410
AMD x86-64 options
AMD 64-bit processors, 408
kernel code model, 409
large code model, 409
-m32, 408
-m64, 408
-mcmodel=kernel, 409
-mcmodel=large, 409
-mcmodel=medium, 409
-mcmodel=small, 409
medium code model, 409
-mno-red-zone, 409
red zone, definition of, 409
small code model, 409
Anderson, Erik, 307

ARC options
ARC 32-bit RISC processor, 411
big endian mode, 411
-EB, 411
-EL, 411
little endian mode, 411
-mcpu=CPU, 411
-mdata=data-section, 411
-mmangle-cpu, 411
-mrodata=readonly-data-section, 411
-mtext=text-section, 411
ARM options
Acorn Archimedes R260, 413, 417
Acorn Computer Systems, 412
ARM Application Binary Interface (ABI), 412
ARM instruction set, 412
ARM Procedure Call Standard (APCS), 412, 415
assembler post-processor, 415
bare metal ARM code, 413
big endian mode, 413
BSD-mode compiler, 413
Cirrus Logic, 413
COFF output toolchains, 416
compiling libgcc.a, 416
interworking mode, 413
-mabi=name, 412
-mabort-on-noreturn, 412
-malignment-traps, 412
-mapcs, 412
-mapcs-26, 412
-mapcs-32, 413
-mapcs-float, 413
-mapcs-frame, 413
-mapcs-reentrant, 413
-mapcs-stack-check, 413
-march=name, 413
-marm, 413
-mbig-endian, 413
-mbsd, 413
-mcallee-super-interworking, 413
-mcaller-super-interworking, 413
-mcirrus-fix-invalid-isns, 413
-mcpu=name, 414
-mfloat-abi=name, 414
-mfp=number, 414
-mfpe=number, 414
-mfpu=name, 414
-mhard-float, 414
-mlittle-endian, 414
-mlong-calls, 414
-mno-alignment-traps, 414
-mno-apcs-frame, 415
-mno-long-calls, 415
-mnop-fun-dllimport, 415
-mno-sched-prolog, 415
-mno-short-load-bytes, 415
-mno-short-load-words, 415
-mno-soft-float, 415
-mno-symrename, 415
-mno-thumb-interwork, 415
-mno-tpcs-frame, 415
-mno-tpcs-leaf-frame, 415
-mpic-register=name, 415
-mpoke-function-name, 415
-msched-prolog, 416
-mshort-load-bytes, 416
-mshort-load-words, 416
-msingle-pic-base, 416
-msoft-float, 416
-mstructure-size-boundary=n, 416
-mthumb, 416
-mthumb-interwork, 416

Find it faster at http://superindex.apress.com

-mtpcs-frame, 416
-mtpcs-leaf-frame, 416
-mtune=name, 417
-mwords-little-endian, 417
-mxopen, 417
NOOPs, 413
PIC addressing, 415–416
RISC iX, 413, 415, 417
Thumb instruction set, 412
Thumb Procedure Call Standard, 415–416
autoconf
approaches to creating a Makefile template, 165
autom4te application, 164
autom4te.cache directory, 164
autoscan and the configure.scan file, 162
autoscan utility, 161
autoscan.log file, contents of, 162
config.log file, contents of, 165
config.status shell script, 170, 172
configure scripts, command-line options, 175
configure scripts, creating, 172
configure scripts, executing, 164–165, 172
configure scripts, --prefix option, 154, 175
configure.ac file, code example, 170
configure.ac file, output from, 164
configure.ac files, creating, 161
configure.scan file, output from, 162
configuring the source code, 156
creating a configuration description file, 161
default installation location, 154
downloading and extracting the gzipped source archive, 155
executing a configure script, 173
function of, 151
home page, 155
installation alternatives to overwriting existing binaries, 154
Libtool, 181, 185, 194
Linux distributions and, 155
m4 macro processor, 152
MacKenzie, David, 152
macro entries in the sample autoconf file, 163
make command, 156
make install command, 157
operation of, 152
Perl scripts and auxiliary utilities, table of, 158
procedure for configuring an application, 169–170
related mailing lists, 155
square brackets and arguments passed to autoconf macros, 164
testing with the make check command, 156–157
which autoconf command, 154
automake
aclocal scripts, 160, 169–171
auxiliary text files, 168
building the sample application from the Makefile, 173–174
commonly used automake primaries, 166
configuring the source code, 159–160
creating a build description file, 161
creating auxiliary files from the automake installation, 171
default installation location, 154
download page, 159
downloading and extracting the gzipped source archive, 159

executing with the --add-missing command-line option, 168
function of, 151
GNU Makefile conventions, 153
home page, 159
ifnames script, 170
installation alternatives to overwriting existing binaries, 154
Libtool, 181, 185, 194
Linux distributions and, 158
MacKenzie, David, 153
make command, 160
make install command, 161
Makefile targets produced by automake, 167
Makefile.am file, 153, 165–166
Makefile.in file, 153, 165, 172
Perl version of, 153
procedural overview for using, 168–169
procedure for configuring an application, 169–170
related mailing lists, 159
requirements for Perl interpreter, 159
testing with the make check command, 160–161
AVR options
AVR instruction set, 417
-mcall-prologues, 417
-mdeb, 417
-minit-stack=n, 417
-mint8, 417
-mmcu=MCU, 417
-mno-interrupts, 418
-mno-tablejump, 418
-mshort-calls, 418
-msize, 418
-mtiny-stack, 418

B

Backus, John, 54
basic block, defined, 124
Berkeley Standard Distribution (BSD), 151, 248, 423
Blackfin options
Analog Devices, 418
Blackfin processors, 418
CSYNC instructions, 418
-mcsync-anomaly, 418
-mid-shared-library, 418
-mlong-calls, 418
-mlow64k, 418
-mno-csync-anomaly, 418
-mno-id-shared-library, 418
-mno-long-calls, 418
-mno-low64k, 418
-mno-omit-leaf-frame-pointer, 418
-mno-specld-anomaly, 419
-momit-leaf-frame-pointer, 419
-mshared-library-id=n, 419
-mspecld-anomaly, 419
SSYNC instructions, 418
buildroot
Altera Nios II processors, 307
AMD 64-bit processors, 308
Anderson, Erik, 307
ARM 32-bit RISC processors, 307
Build Options screen, 311
building a uClibc-based cross-compiler, 309–310, 312, 314, 316
Code Reduced Instruction Set processors, 307
creating basic root filesystems that use BusyBox, 307
debugging toolchain build problems, 317
DEC 64-bit RISC processors, 307

Find it faster at http://superindex.apress.com

downloading and extracting, 308
function of, 308
Hitachi SuperH 32-bit RISC
processors, 308
home page of the buildroot project, 307
Intel 32-bit processors, 307
list of supported platforms, 307–308
manually reconfiguring uClibc, 317
manually resetting buildroot and uClibc configuration values, 317
MIPS 32-bit processors, 307
Motorola 680x0 CISC processors, 307
PowerPC 32-bit RISC processors, 307
rerunning the make distclean command, 317
resolving uClibc-related configuration issues, 317
specifying the installation directory, 309
Subversion Source Code Control System (SCCS), 308
Sun Microsystems 32-bit RISC processors, 308
Target Architecture Variant configuration option, 310
Target Options configuration screen, 314
Toolchain Options configuration screen, 312
Burley, James Craig, 55, 74
Burrows-Wheeler compression algorithm, 259
BusyBox utility, 203, 266, 270–271, 307

C

call graph
call arc, 123
defined, 123
.gcno file, 123–124
Clipper options
Clipper family of RISC processors, 419
Intergraph Unix workstations, 419
-mc300, 419
-mc400, 419
code coverage analysis
cross-language support, 120
defined, 119
code libraries
definition of, 177
dynamic link libraries (DLLs), 181
dynamically loaded (DL) libraries, using, 180–181
GNU Glibc library, 181
ldconfig application, 179, 188
libltdl.so library, 181
linker name, 180
Pluggable Authentication Modules (PAM), 180
shared libraries, advantages and disadvantages, 178
shared libraries, naming conventions, 179
shared libraries, version numbering scheme, 180
shared library loader, 179
soname, 180
source code modules and linked libraries, 177
static libraries, advantages and disadvantages, 178
static libraries, extensions, 177
code profiling
cross-language support, 120
defined, 119
execution profiling, 119

compiler optimization theory
algorithm improvement, 102
basic block, definition of, 102
code inlining, 105
code motion, 103
common subexpression elimination (CSE), 103, 110
constant folding, 103
constant propagation, 104
control flow analysis, 102
copy propagation transformations, 104
data flow analysis, 102
dead code elimination (DCE), 104–105
function inlining, 105
global transformations, 102
if-conversion, 105
inlining, 105
local transformations, 102
loop unrolling, 105
loop-invariant code motion, 103
optimization and debugging, 101
optimization, purpose of, 102
optimizing compilers, 102
partial redundancy elimination, 103
producing optimized code, 102
transformation vs. optimization techniques, 102
transformations on intermediate code representations, 102
unreachable code elimination, 104
compiling GCC from source
advantages of doing it yourself, 227
autoconf, 229
automake, 229
bash, 228
Bison, 228
Boehm-Demers-Weiser conservative garbage collector, 239
build system, definition of, 233
building GCC from the Subversion tree, 229
building the GCC Fortran compiler, 239
building the GCC Java compiler, 239
building the GCC Objective-C compilers, 239
building the test harness, 243
canadian compiler, 233
Canadian Cross build, 233
configuring the source code, 232
crossback compiler, 233
cross-compiler, 233
crossed native compiler, 233
customizing a build for your system, 227
DejaGNU, 228, 242
disk space requirements, 229–230
disregarding warning messages during the build process, 240
distinguishing the source directory from the build tree, 232
downloading the source code, 231
Expect, 229, 242
extracting the tarballs, 231
Flex, 228
gcc, 228
GCC as either a primary or secondary compiler, 227
GCC Makefile targets, 241–242
GCC test suite result codes, 244
general procedure for building and installing utilities, 229
Gettext, 229
GMP (GNU Multiple Precision Arithmetic Library), 229
GNAT, 228

Find it faster at http://superindex.apress.com

GNU binutils, 228
GNU make, 229
GNU tar, 229
Gperf, 229
host system, definition of, 233
installing GCC into a subdirectory of /usr/local, 230–231
installing the source code, 231–232
invoking the configure script, 233
keeping multiple GCC compilers on your system, 233
keeping the source and build directories distinct, 232
ldconfig command, 245
make bootstrap command, 239
make install command, 245
MPFR (multiple-precision floating-point rounding), 229
native compiler, 233
nice command, 240
NLS-related configuration options, 239
options for GCC's configure script, 234–238
overview of the build process, 228
participating in GCC development, 228
performing a three-stage build of the compiler, 240
possible combinations of build, host, and target systems, 233
recommendations for building select languages, 232
reconfiguring GCC by running the make distclean command, 228
renice command, 240
required libraries for installing the GCC Fortran compiler, 229
root vs. nonroot privileges during installation, 231
running subsets of the GCC test suite, 243
running the GCC test suite, 242–244
starting a parallel build on an SMP system, 240
steps executed in a full bootstrap build, 240–241
target system, definition of, 233
Tcl (Tool Command Language), 229, 242
test harness, definition of, 243
using Glibc 2.2.3 with GCC 3.0 or later, 230
using the test_summary script, 244
verifying required utilities, 228–230
which expect command, 242
which tcl command, 242
Comprehensive Perl Archive Network, 257
Convex options
Convex Computer, 419
Cray Research, 419
Exemplar systems, 419
-margcount, 419
-mc1, 420
-mc2, 420
-mc32, 420
-mc34, 420
-mc38, 420
-mlong32, 420
-mlong64, 420
-mnoargcount, 420
-mvolatile-cache, 420
-mvolatile-nocache, 420
CRIS options
Axis Solutions, 420
Code Reduced Instruction Set (CRIS) processors, 420
ETRAX 100, 421–422
GOT (global offset table), 421
-m16-bit, 420

-m32-bit, 420
-m8-bit, 420
-maout, 420
-march=architecture-type, 421
-mbest-lib-options, 421
-mcc-init, 421
-mconst-align, 421
-mcpu=architecture-type, 421
-mdata-align, 421
-melf, 421
-melinux, 421
-melinux-stacksize=n, 421
-metrax100, 421
-metrax4, 421
-mgotplt, 421
-mlinux, 421
-mmax-stack-frame=n, 421
-mmul-bug-workaround, 421
-mno-const-align, 421
-mno-data-align, 421
-mno-gotplt, 422
-mno-mul-bug-workaround, 422
-mno-prologue-epilogue, 422
-mno-side-effects, 422
-mno-stack-align, 422
-moverride-best-lib-options, 422
-mpdebug, 422
-mprologue-epilogue, 422
-mstack-align, 422
-mtune= architecture-type, 422
PLT (procedure linkage table), 421
-sim, 422
-sim2, 422
cross-compilers
advantages of GCC as a cross-compiler, 299
building cross-compilers manually, 300, 318–320
buildroot, 300
checking Web sites related to your architecture, 319
conventions for GCC prefixes, 300
cross-compilation targets supported by GCC, 300
cross-compilation, definition of, 299
crossdev project, 300
crosstool, 300
DESTenvironment variable, 319
difficulties in building cross-compilers, 299
ELDK (Embedded Linux Development Kit), 300
embedded Linux systems and, 299
host and target systems, 299
open source tools for building cross-compilers, 300
required source packages for building a cross-compiler, 318
TARGET environment variable, 319
crosstool
AMD 64-bit processors, 303
applying a patch to a component software package, 306
ARM 32-bit RISC processors, 301
creating a package configuration file, 303
creating a platform configuration file, 306
crossgcc, 300
Cygwin Linux emulation environment for Windows, 302
DEC 64-bit RISC processors, 301
downloading and extracting, 304
environment variables and, 301
environment variables needing to be set manually, 303
executing the all.sh script, 304, 306
Gatliff, Bill, 300
GCC_LANGUAGES environment variable, 303

Find it faster at http://superindex.apress.com

Hitachi 32-bit RISC SuperH processors, 303
IBM 64-bit processors, 303
information resources on, 307
Intel 64-bit processors, 302
Intel Pentium 4 processors, 303
Intel Wireless MMX technology, 301
Itanium processors, 302
Kegel, Dan, 300
list of platform configuration files and supported platforms, 301–303
matrix of platforms and package versions, 303
MIPS 32-bit processors, 302
Motorola 680x0 CISC processors, 302
operation of, 301
package configuration file, 301
Pentium-class IA-32 processor, 302
PowerPC 32-bit RISC processors, 302
PowerPC 64-bit RISC G5 processors, 302
procedure for building a custom cross-compiler, 305
procedure for building a default cross-compiler, 304–305
RESULT_TOP environment variable, 303
Sun Microsystems 32-bit RISC processors, 303
Sun Microsystems 64-bit RISC processors, 303
TARBALLS_DIR environment variable, 303
updating a package configuration file, 306
CRX options
-mloopnesting=n, 423
-mmac, 423
-mno-push-args, 423
-mpush-args, 423

D

D30V options
D30V RISC processor, 423
-masm-optimize, 423
-mbranch-cost=n, 423
-mcond-exec=n, 423
-mextmem, 423
-mextmemory, 423
-mno-asm-optimize, 423
-monchip, 423
real-time MPEG-2 decoder, 423
Darwin options
-all_load, 424
Aqua graphical components, 424
-arch_errors_fatal, 424
Berkeley Standard Distribution (BSD), 423
-bind_at_load, 424
-bundle, 424
-bundle_loader executable, 424
-dynamiclib, 424
-Fdir, 424
-ffix-and-continue, 425
-findirect-data, 425
-force_cpusubtype_ALL, 424
-gfull, 424
-gused, 424
Mach operating system, 423
Mach-O bundle format file, 424
Macintosh OS X operating system, 423
-mfix-and-continue, 425
-mmacosx-version-min=version, 425
-mone-byte-bool, 425
OS X fat (multiarchitecture) binaries, 424
STABS debugging format, 424
Xcode development environment, 424

DEC Unix, 405
DES cryptographic functions, 254
dietlibc
  adding the –v switch to produce verbose output, 285
  building, 284
  cross-compiling, 284
  diet front-end driver application, 282, 285
  features of, 283
  gcc driver program, 283
  installing, 284
  licensing under the GNU General Public License (GPL), 283
  Makefile for, 284
  naming convention for GCC-based cross-compilers, 284
  obtaining and retrieving, 284
  platforms used on, 283
  primary use as a static (not shared) library, 283
  using with a cross-compiler, code example, 285
  using with gcc, 285

E

Embedded Linux Development Kit (ELDK), 300

F

f2c Fortran-to-C conversion utility
  support for ANSI FORTRAN 77, 76
  use with g77 and gfortran, 76
Feldman, Stu, 153
Fibonacci sequence
  compiling the Fibonacci.class file, 85
  fibonacci.c sample application, source code, 126–127
  fibonacci.c.gcov output file, 128–129
  Fibonacci.java, code example, 83–84
  sample code modernized, 59–61
  sample code, 57
floating-point control register (FPCR), 405
floating-point register (FPR), 405
Fortran
  advantages of, 53
  Backus, John, 54
  f2c Fortran-to-C conversion utility, 76
  FAQ for, 53
  Fortran 90, 55, 57–58, 64, 75, 77
  Fortran 95, 55, 57, 61, 64, 75
  g77 compiler, 55, 74–75
  g95 compiler, 76–77
  gfortran compiler, 55–59, 61–74, 77
  history and development, 54–55
  information resources, 77
  Intel's Fortran compiler, 76
Free Software Foundation (FSF), 152–153, 155, 159
  contacting GNU Press, 225
  tar utility, 258
Free Standards Group
  open standards for internationalization under Linux, 264
FR-V options
  FDPIC ABI, 426
  GPREL (global pointer relative) relocations, 426
  -macc-4, 425
  -macc-8, 425
  -malign-labels, 426
  -malloc-cc, 426
  -mcond-exec, 426
  -mcond-move, 426
  -mcpu=CPU, 426
  -mdouble, 426

Find it faster at http://superindex.apress.com

-mdword, 426
-mfdpic, 426
-mfixed-cc, 426
-mfpr-32, 426
-mfpr-64, 426
-mgpr-32, 426
-mgpr-64, 426
-mgprel-ro, 426
-mhard-float, 426
-minline-plt, 427
-mlibrary-pic, 427
-mlinked-fp, 427
-mlong-calls, 427
-mmedia, 427
-mmuladd, 427
-mmulti-cond-exec, 427
-mnested-cond-exec, 427
-mno-cond-exec, 427
-mno-cond-move, 427
-mno-double, 427
-mno-dword, 427
-mno-eflags, 427
-mno-media, 427
-mno-muladd, 427
-mno-multi-cond-exec, 427
-mno-nested-cond-exec, 428
-mno-optimize-membar, 428
-mno-pack, 428
-mno-scc, 428
-mno-vliw-branch, 428
-moptimize-membar, 428
-mpack, 428
-mscc, 428
-msoft-float, 428
-mTLS, 428
-mtls, 428
-mtomcat-stats, 428
-multilib-library-pic, 428
-mvliw-branch, 428
PLT (procedure linkage table) entries, 427
Very Long Instruction Word (VLIW) processor, 425

G

g++ C++ compiler
__attribute__ attribute, code example, 51
Borland template model, 49
-c option, code example, 42
c++ input files, 43
c++-cpp-output input files, 43
Cfront (AT&T) template model, 49
combining C++ and Java code, 48
compiling a single source file, 41
compiling multiple source files, 42
conformance to the application binary interface (ABI), 46–47
ELF symbols, exporting of, 51
extending an existing ABI through versioning, 47
extern keyword, 49
-fabi-version=n option, 47
-fhidden option, 51
-fno-implicit-templates option, 49
-frepo option, 49
__FUNCTION__ identifier, 49
function name identifiers in C++ and C, 49–50
-fvisibility=value option, 51
g++ --version command, 47
GCC filename suffixes, 43
GCC specs file, 42
handling a filename with no recognized suffix, 43
identifying Java exception handling, 50
init_priority attribute, 48
inline keyword, 49

java_interface attribute, 48
language extensions, 47
making selected symbols visible, 51
minimum and maximum value operators, 50
mixing the order of options and compiler arguments, 41
-o option, code example, 42
__PRETTY_FUNCTION__ identifier, 49
single-letter and multiletter grouping options, 41
standard command-line options, 43–45
static keyword, 49
template instantiation, 49
templates, functions of, 49
using g++ to invoke GCC's C++ compiler directly, 43
versioning, definition of, 47
visibility attributes and pragmas, 51–52
visibility pragma, code example, 52
warning-related options, 45–46
-x lang option, 43
g77 compiler
acceptance of non-FORTRAN 77 capabilities, 75
Burley, James Craig, 55, 74
calling conventions, 76
case-sensitivity in, 74
compared to gfortran compiler, 74–75
documentation for GCC 3.4.5, 74
entity names, 75
-fvxt option, 75
interoperability with other Fortran compilers, 75
Mac OS X version, 74
-ugly options, 75
widespread use and stability of, 74
Gatliff, Bill, 300
gcc C compiler
alternate keywords, 7
ANSI C standard (C89), 3
-ansi option, 6
bit fields, signed and unsigned, 5
built-in functions, definition of, 6
C dialect command-line options, 3–5
-c option, code example, 2
C99 standard, 3, 6
char type, signed and unsigned, 5
compiling a single source file, 2
compiling C dialects, 3–7
compiling multiple source files, 2
effects of turning on standards compliance, 6–7
-fno-builtin option, 6
freestanding (unhosted) environment, definition of, 4, 6
GCC specs file, 3
handling unused objects and unreachable code, 10
hosted environment, definition of, 4–5
ISO/ANSI C, 3
mixing the order of options and compiler arguments, 1
-o option, code example, 2
-pedantic option, 6–7
-pedantic option, using with -Wformat, 8
-pedantic-errors option, 6
pre-ISO C, 6
preventing format string exploits, 7, 9
printme.c application, code examples, 8–9
program with an unused variable, code example, 10
single-letter and multiletter grouping options, 1
-std=c89 option, 6
-std=c99 option, 6

Find it faster at http://superindex.apress.com

translation unit, definition of, 10
verifying code adherence to various standards, 3
warning message, definition of, 7
-Wformat option, 7
-Wformat-security option, 9
-Wno-format-extra-args option, 9
-Wunreachable-code option, 10
-Wunused option, 9–10
GCC compilers
-###, 324
adding debugging information, 343
ar utility, 336
.as filename extension, 329
boosting compilation speed with a RAM disk, 349
built-in spec strings, table of, 351–352
-c, 328–329
C_INCLUDE_PATH, 348
catching unused objects and unreachable code, 342
command-line output options, table of, 326
commonly used debugging options, table of, 343–344
compilation phase errors, 324
COMPILER_PATH, 347
compiling single or multiple source files, 327
controlling the linker, 335
controlling the preprocessor, 331
CPATH, 348
CPLUS_INCLUDE_PATH, 348
customizing with environment variables, 347
-D, 332
-dCv, 347
defining the name of an output file, 327
demonstrating the equivalence of building and linking code, 337
DEPENDENCIES_OUTPUT, 348
differentiating -lname options and specified object filenames, 336
-dmod (dump option), arguments for, 345–346
-dumpmachine, 323–324
-dumpspecs, 350
-dumpversion, 323–324
-dv, 347
-E, 328–331
--eh-frame-hdr, 338
enabling and disabling preprocessor macros, 332
examining a compiler's assembly level output, 330
filename suffixes and compilation types, table of, 324–326
forcing input files to be treated as source code files, 328
four compilation stages, 321
-g, 343–345
GCC_EXEC_PREFIX, 347–348
GCC_EXEC_PREFIX, operation under Cygwin, 348
general GCC command-line options, table of, 322–323
general GCC warning options, 339–342
-ggdb, 343, 345
grouping multiple single-letter options, 321
halting compilation after preprocessing, 330
hand-tuning assembly code, 330
header file search path, 333
-I, 334, 348
-I dir, 333

.i filename extension, 329
#if 0...#endif construct, 333
-imacros file, 332
interprocess communication (IPC) mechanisms, 324
-iquote, 334
-isystem, 348
keeping the functions defined in the libgcc.a library, 338
-L dir, 334, 336
LANG, 349
@language spec string, 351
LC_ALL, 349
LC_CTYPE, 349
LC_MESSAGES, 349
libraries as archive files, 336
library directory search list, 334
library search patch, 336
LIBRARY_PATH, 349
link options, table of, 335
linking, definition of, 335
Linux tmpfs filesystem, 349
-lname, 336–337
long name options with positive and negative forms, 322
-M, 348
make utility, 330, 348
-MF, 348
mixing the order of options and arguments, 322
-MM, 348
modifying directory search paths, 333
-MT, 348
multilibs, definition of, 388
#name spec string, 351
-nodefaultlibs, 337
-nostartfiles, 337
-nostdlib, 338
-o, 327, 329
OBJC_INCLUDE_PATH, 348
object files, 324, 326
options for modifying directory search paths, 333
-pass-exit-codes, 324
performing a compilation one step at a time, 329–330
-pipe, 324
.pre filename extension, 329
predefined substitution specs, 352–353
preprocessor magic (abuse), 330
preprocessor options, table of, 331
-print-, 324
-S, 328–330
-shared-libgcc, 338
single-letter and multiletter options, 321
spec file commands, table of, 350
spec file directives, table of, 350
spec file suffix rules, 351
spec file, definition of, 349
spec processing instructions, table of, 353–354
spec strings, 323, 335, 349
specifying multiple -I dir options, 334
specifying multiple opt options, 335, 338
specifying multiple spec files, 350
-specs=file, 350
standard system include directories, 333
-static-libgcc, 338
stopping compilation after preprocessing, 328
substitution, 353
support for static and shared libraries, 338
system header files, 334
system time, 390
-time, 324
TMPDIR, 349

Find it faster at http://superindex.apress.com

-U, 332
-Uname, 332
user header files, 334
user time, 390
using preprocessor macros, code example, 332
using several options of the same kind, 322
using shared libgcc vs. static libgcc, 338
--version, 324
Visualization of Compiler Graphs (VCG), 347
-Wa,opt, 335
-Wall, 343
warning message, definition of, 338
-Werror, 343
-Wl,opt, 338
-Wunreachable-code, 342
-Wunused, 342
-x, 328, 329
-x assembler, 330
-x cpp-output, 329
-x lang, 326
GCC machine-dependent options
cross-compiler usage and, 403
location of Darwin support, 403
location of GCC configuration information, 403
-m command-line option, 403
GCC machine-independent options
-###, 355
-A-, 355
-ansi, 355
-AQUESTION=ANSWER, 355
-aux-info filename, 355
-b machine, 355
-Bprefix, 355
-C, 356
-c, 356
-CC, 356
-combine, 356
-dletters, 356–357
-DMACRO, 356
-dumpmachine, 357
-dumpspecs, 357
-dumpversion, 357
-E, 357
-fabi-version=n, 357
-falign-functions, 357
-falign-jumps, 358
-falign-labels, 358
-falign-loops, 358
-fallow-single-precision, 358
-falt-external-templates, 358
-fargument-alias, 358
-fasynchronous-unwind-tables, 358
-fbounds-check, 358
-fbranch-probabilities, 358, 373, 380
-fbranch-target-load-optimize, 358
-fbranch-target-load-optimize2, 359
-fbtr-bb-exclusive, 359
-fbuiltin, 365
-fcaller-saves, 359
-fcall-saved-reg, 359
-fcall-used-reg, 359
-fcheck-new, 359
-fcond-mismatch, 359
-fconserve-space, 359
-fconstant-string-class=classname, 359
-fcprop-registers, 359
-fcross-jumping, 359
-fcse-follow-jumps, 360
-fcse-skip-blocks, 360
-fcx-limited-range, 360
-fdata-sections, 360
-fdelayed-branch, 360, 385

-fdelete-null-pointer-checks, 360
-fdiagnostics-show-location, 360
-fdiagnostics-show-options, 360
-fdollars-in-identifiers, 360
-fdump-class-hierarchy, 360
-fdump-ipa-switch, 361
-fdump-translation-unit, 361
-fdump-tree-switch, 361–362
-fdump-unnumbered, 362
-fearly-inlining, 362
-feliminate-dwarf2-dups, 362
-feliminate-unused-debug-symbols, 362
-feliminate-unused-debug-types, 363
-femit-class-debug-always, 363
-fexceptions, 363, 379
-fexpensive-optimizations, 363
-fexternal-templates, 358, 363
-ffast-math, 363
-ffinite-math-only, 363
-ffixed-reg, 363
-ffloat-store, 363
-fforce-addr, 364
-fforce-mem, 364
-ffor-scope, 364, 368
-ffreestanding, 364
-ffriend-injection, 364
-ffunction-sections, 364, 380
-fgcse, 364
-fgcse-after-reload, 364
-fgcse-las, 364
-fgcse-lm, 365
-fgcse-sm, 365
-fgnu-runtime, 365
-fhosted, 365
-fif-conversion, 365
-finhibit-size-directive, 365
-finline-functions, 365, 386
-finline-functions-called-once, 365
-finline-limit=n, 365
-finstrument-functions, 365
-fipa-pta, 366
-fivopts, 366
-fkeep-inline-functions, 366
-fkeep-static-consts, 366
-fleading-underscore, 366
-fmath-errno, 366
-fmem-report, 366
-fmerge-all-constants, 366
-fmerge-constants, 366
-fmessage-length=n, 366
-fmodulo-sched, 366
-fmove-all-movables, 367
-fmove-loop-invariants, 367
-fms-extensions, 367
-fmudflapir, 367
-fmudflapth, 367
-fnext-runtime, 367
-fno-access-control, 367
-fno-asm, 367
-fno-branch-count-reg, 367
-fno-builtin, 367
-fno-common, 368
-fno-const-strings, 368
-fno-cprop-registers, 368
-fno-default-inline, 368
-fno-defer-pop, 368
-fno-elide-constructors, 368
-fno-enforce-eh-specs, 368
-fno-for-scope, 368
-fno-freestanding, 365
-fno-function-cse, 368
-fno-gnu-keywords, 369
-fno-gnu-linker, 369
-fno-guess-branch-probability, 369

Find it faster at http://superindex.apress.com

-fno-ident, 369
-fno-implement-inlines, 369
-fno-implicit-inline-templates, 369
-fno-implicit-templates, 369, 374
-fno-inline, 369
-fno-jump-tables, 369
-fno-math-errno, 369
-fnon-call-exceptions, 371
-fno-nil-receivers, 369
-fno-nonansi-builtins, 370
-fno-operator-names, 370
-fno-optional-diags, 370
-fno-peephole, 370
-fno-reschedule-modulo-scheduled-loops, 374
-fno-rtti, 370
-fno-sched-interblock, 370
-fno-sched-spec, 370
-fno-signed-bitfields, 370
-fno-stack-limit, 370
-fno-threadsafe-statics, 370
-fno-toplevel-reorder, 370
-fno-trapping-math, 370
-fno-unsigned-bitfields, 371
-fno-verbose-asm, 371
-fno-weak, 371
-fno-zero-initialized-in-bss, 371
-fobjc-call-cxx-cdtors, 371
-fobjc-direct-dispatch, 371
-fobjc-exceptions, 371
-fobjc-gc, 371
-fomit-frame-pointer, 371
-fopenmp, 371
-foptimize-register-move, 371, 373
-foptimize-sibling-calls, 371
-fpack-struct, 372
-fpcc-struct-return, 372–373
-fpeel-loops, 372–373
-fpermissive, 372
-fpic, 372, 377, 389
-fPIE, 372
-fprefetch-loop-arrays, 372
-fpretend-float, 372
-fprofile-arcs, 369, 373, 377, 380
-fprofile-generate, 373
-fprofile-use, 373
-fprofile-values, 373
-frandom-seed=STRING, 373
-freduce-all-givs, 373
-fregmove, 373
-freg-struct-return, 373
-frename-registers, 373, 386
-freorder-blocks, 373
-freorder-blocks-and-partition, 373
-freorder-functions, 373
-freplace-objc-classes, 374
-frepo, 374
-frerun-cse-after-loop, 374, 379
-frerun-loop-opt, 374
-freschedule-modulo-scheduled-loops, 374
-frounding-math, 374
-frtl-abstract-sequences, 374
-fsched2-use-superblocks, 374–375
-fsched2-use-traces, 375
-fsched-spec-load, 374
-fsched-spec-load-dangerous, 374
-fsched-stalled-insns, 374
-fsched-stalled-insns-dep=n, 374
-fschedule-ins, 375
-fschedule-insns, 370, 374–375
-fschedule-isns2, 374
-fsched-verbose=n, 374
-fsection-anchors, 375
-fshared-data, 375

-fshort-double, 375
-fshort-enums, 375
-fshort-wchar, 375
-fsignaling-nans, 375
-fsigned-bitfields, 375
-fsigned-char, 375
-fsingle-precision-constant, 375
-fsplit-ivs-in-unroller, 375
-fssa, 376
-fssa-ccp, 376
-fssa-dce, 376
-fstack-check, 376
-fstack-limit-register=reg, 376
-fstack-limit-symbol=SYM, 376
-fstack-protect, 399
-fstack-protector, 376
-fstack-protector-all, 376
-fstats, 376
-fstrength-reduce, 376, 379
-fstrict-aliasing, 376
-fstrict-warning, 399
-fsyntax-only, 376
-ftemplate-depth-n, 376
-ftest-coverage, 373, 377
-fthread-jumps, 377, 385
-ftime-report, 377
-ftls-model=MODEL, 377
-ftracer, 373, 377
-ftrapping-math, 375
-ftrapv, 377
-ftree-ccp, 377
-ftree-ch, 377
-ftree-copy-prop, 377
-ftree-copyrename, 377
-ftree-dce, 377
-ftree-dominator-opts, 377
-ftree-dse, 377
-ftree-fre, 377
-ftree-loop-im, 378
-ftree-loop-ivcanon, 378
-ftree-loop-linear, 378
-ftree-loop-optimize, 378
-ftree-lrs, 378
-ftree-pre, 378
-ftree-salias, 378
-ftree-sink, 378
-ftree-sra, 378
-ftree-store-ccp, 378
-ftree-store-copy-prop, 378
-ftree-ter, 378
-ftree-vect-loop-version, 378
-ftree-vectorize, 378
-ftree-vectorizer-verbose=n, 378
-funroll-all-loops, 378
-funroll-loops, 373, 379–380, 386
-funsafe-loop-optimizations, 379
-funsafe-math-optimizations, 379
-funsigned-bitfields, 379
-funsigned-char, 379
-funswitch-loops, 379
-funwind-tables, 379
-fuse-cxa-atexit, 379
-fvariable-expansion-in-unroller, 379
-fvar-tracking, 379
-fverbose-asm, 379
-fvisibility=VALUE, 380
-fvisibility-inlines-hidden, 380
-fvolatile, 380
-fvolatile-global, 380
-fvolatile-static, 380
-fvpt, 373, 380
-fvtable-gc, 380
-fweb, 380
-fwhole-program, 380

Find it faster at http://superindex.apress.com

-fworking-directory, 380
-fwrapv, 380
-fwritable-strings, 368, 380
-fzero-link, 381
-g, 381
-gcoff, 381
-gdwarf, 381
-gen-decls, 381
-ggdb, 381
-gLEVEL, 381
-gstabs, 382
-gvms, 382
-gxcoff, 382
-gxcoff+, 382
-H, 382
--help, 382
-I-, 382
-IDIR, 383
-idirafter, 384
-imacros file, 383
-imultilib dir, 383
-include file, 383
-iprefix prefix, 383
-iquote dir, 383
-isysroot dir, 383
-isystem, 383
-isystem dir, 383
-iwithprefix dir, 384
-iwithprefixbefore dir, 384
-Ldir, 384
-lLIBRARY, 384
-M, 384
-MD, 384
-MF file, 384
-MG, 385
-MM, 385
-MMD, 385
-MP, 385
-MQ target, 385
-MT target, 385
-nodefaultlibs, 385
-no-integrated-cpp, 385
-nostartfiles, 385
-nostdinc, 385
-nostdinc++, 385
-nostdlib, 385
-O, 385
-O0, 386
-O2, 386
-O3, 386
-Os, 377–378, 386
-P, 386
--param, 386–387
-pass-exit-codes, 387
-pedantic, 372, 387, 398
-pedantic-errors, 387
-pg, 387
-pie, 387
-pipe, 388
-print-file-name=LIBRARY, 388
-print-libgcc-file-name, 388
-print-multi-directory, 388
-print-multi-lib, 388
-print-prog-name=PROGRAM, 389
-print-search-dirs, 389
-Q, 389
-rdynamic, 389
-remap, 389
-S, 389
-save-temps, 356, 389
-shared, 389
-shared-libgcc, 389
-specs=file, 389
-static, 389

-static-libgcc, 389
-std=std, 390
-symbolic, 390
--sysroot=dir, 390
--target-help, 390
-time, 390
-traditional, 390–391
-traditional-cpp, 391
-trigraphs, 391
-UNAME, 391
-undef, 391
-v, 391
-V VERSION, 391
--version, 392
-W, 392, 399
-w, 401
-Wa,option, 392
-Wabi, 392
-Waggregate-return, 392
-Wall, 392, 398
-Walways-true, 392
-Wassign-intercept, 392
-Wbad-function-cast, 392
-Wc++-compat, 392
-Wcast-align, 392
-Wcast-qual, 392
-Wchar-subscripts, 393
-Wcomment, 393
-Wconversion, 393, 400
-Wctor-dtor-privacy, 393
-Wdeclaration-after-statement, 393
-Wdisabled-optimization, 393
-Wdiv-by-zero, 393
-Weffc++, 393
-Werror, 394
-Werror-implicit-function-declaration, 394
-Wextra, 392, 394
-Wfatal-errors, 394
-Wfloat-equal, 394
-Wformat, 395–397
-Wformat-nonliteral, 395
-Wformat-security, 395
-Wformat-y2k, 395
-Wimplicit, 395
-Wimplicit-function-declaration, 395
-Wimplicit-int, 395
-Wimport, 395
-Winit-self, 395
-Winline, 395
-Winvalid-pch, 395
-Wl,option, 395
-Wlarger-than-len, 395
-Wlong-long, 395
-Wmain, 396
-Wmissing-braces, 396
-Wmissing-declarations, 396
-Wmissing-field-initializers, 396
-Wmissing-format-attribute, 396
-Wmissing-include-dirs, 396
-Wmissing-noreturn, 396
-Wmissing-prototypes, 396
-Wmultichar, 396
-Wnested-externs, 396
-Wno-deprecated, 396
-Wno-deprecated-declarations, 396
-Wno-div-by-zero, 393, 396
-Wno-endif-labels, 396
-Wno-format-extra-args, 397
-Wno-format-y2k, 397
-Wno-import, 397
-Wno-int-to-pointer-cast, 397
-Wno-invalid-offsetof, 397
-Wno-long-long, 395
-Wno-multichar, 396–397

Find it faster at http://superindex.apress.com

-Wnonnull, 398
-Wno-non-template-friend, 397
-Wnon-template-friend, 398
-Wnon-virtual-dtor, 398
-Wno-pmf-conversions, 397
-Wno-pointer-to-int-cast, 397
-Wno-pragmas, 397
-Wno-protocol, 397
-Wno-return-type, 397, 399
-Wno-sign-compare, 397
-Wold-style-cast, 398
-Wold-style-definition, 398
-Woverlength-strings, 398
-Woverloaded-virtual, 398
-Wp,option, 398
-Wpacked, 398
-Wpadded, 398
-Wparentheses, 398
-Wpointer-arith, 398
-Wpointer-sign, 398
-Wredundant-decls, 399
-Wreorder, 399
-Wreturn-type, 399
-Wselector, 399, 401
-Wsequence-point, 399
-Wshadow, 399
-Wsign-compare, 399
-Wsign-promo, 399
-Wstack-protector, 399
-Wstrict-aliasing, 399
-Wstrict-null-sentinel, 399
-Wstrict-prototypes, 399
-Wswitch, 400
-Wswitch-default, 400
-Wswitch-enum, 400
-Wsynth, 400
-Wsystem-headers, 400
-Wtraditional, 400
-Wtrigraphs, 400
-Wundeclared-selector, 401
-Wundef, 401
-Wuninitialized, 395, 401
-Wunknown-pragmas, 400–401
-Wunreachable-code, 401
-Wunsafe-loop-optimizations, 401
-Wunused, 401
-Wunused-function, 401
-Wunused-label, 401
-Wunused-parameter, 401
-Wunused-value, 401
-Wunused-variable, 401
-Wvariadic-macros, 401
-Wvolatile-register-var, 401
-Wwrite-strings, 401
-x, 402
-Xassembler option, 402
-Xlinker option, 402
-Xpreprocessor option, 402
gcj Java compiler
Ahead-of-Time (AOT) compilation, 79
Apache project's Ant utility, 83
build.xml configuration files, 83
-c option, 82
calling Java from C++ main code, 99
classes stored in libgcj.jar, 89
CLASSPATH environment variable, 89, 93
code-generation and optimization options, 88–89
command-line options, 86–87
compilation search order for files/directories, 90
Compiled Native Interface (CNI), 98
compiling bytecode class files, 82

compiling existing class and jar files into executables, 83
compiling hello.java into an executable, 80
compiling multiple source files, 82
compiling the Fibonacci.class file, 85
conformance to the Java ABI, 93
constructing the Java Classpath, 89–90
core class libraries and the GNU Classpath project, 79
creating a Manifest.txt file, 91–92
creating a shared library from a jar file, 92
downloading the J2SE, 84
error messages for main methods, 81
Excelsior JET AOT compiler, 80
executing the javac-generated class files, 85
Fibonacci.java, code example, 83–84
GCC filename suffixes for Java, 86
GCC specs file, 83
GCJ_PROPERTIES environment variable, uses of, 93
gcj-dbtool command, 92
gij command-line options, 96–98
gij GNU interpreter for Java, 79, 94, 96
hello, world application, code example, 80
HotSpot compilers, 79
identifying alternate locations for class or Java source files, 89
jar files, creating and using, 90–92
jar utility and Unix/Linux tar application, 90
Java and C++ integration, 98
Java Virtual Machine (JVM), 79
jcf-dump, command-line options, 94–95
jcf-dump, sample output, 95–96
Just-in-Time (JIT) compilers, 79
jv-scan, command-line options, 94
jv-scan, uses of, 94
-lgij command, 96
mixing the order of options and compiler arguments, 80
-o option, 82
platform-specific object code, 79
single-letter and multiletter grouping options, 80
system-independent Java bytecode, 79
using bytecode as an intermediate format, 82
using javac, Sun's Java compiler, 84
using the -x lang option for input files, 86
warning-related options, 88
gcov
.gcda file, 124, 128, 133
.gcno file, 127
.gcno file and call graphs, 123–124
.gcno file and program flow arcs, 133
.gcov file, 124
analyzing the test run's execution path, 131
application error messages, 128
calc_fib.c auxiliary function, source code, 127
call arc, 123
call graph, automatic generation of, 120
call graph, defined, 123
command-line options, table of, 124–126
displaying absolute branch counts in a test run, 131–132
displaying coverage and summary profiling information, 128
displaying function call summary information, 132
executing gcov with the -b option, 129–130
fibonacci.c sample application, source code, 126–127
fibonacci.c.gcov output file, 128–129
-fprofile-arcs option, 123–124, 129
-ftest-coverage option, 123–124, 129

Find it faster at http://superindex.apress.com

manual commands for enabling code coverage, 127
measuring branch counts absolutely vs. as percentages, 131
output file produced by gcov -b, 130–131
producing an annotated source file, 124
referencing the same include file in multiple source files, 132
requirements for using, 123
running gcov with the -f option, 132
running gcov with the -l option, 132
running the sample application, 127
gfortran compiler
$ gfortran compilation command, 57
case-insensitivity of, 61
code-generation options, 62–63
command-line options, 62
common compilation options, 55
compared to g77 compiler, 74–75
compiling Fortran code, 57–59
compliance with Fortran 95, 55
debugging options, 63
directory-search options, 63
entity names, 75
-ff2c option, 62, 76
-ffree-form option, 59, 64, 75
Fibonacci sequence, sample code, 57
Fibonacci sequence, sample code modernized, 59–61
Fortran version and file extensions, 58, 75
Fortran-dialect options, 63–64
-funderscoring option, 75
GCC 4.1, 65
getarg() function, 60–61
GNU Fortran standard, conforming to, 61
--help command output, 56
iargc() function, 60–61
information sources, 77
interoperability with other Fortran compilers, 75
intrinsic functions and extensions, 65–74
-o filename option, 58
-std=f95 option, 61, 64
warning options, 64–65
Glibc (GNU C library)
ANSI C, 248
backing out of an unsuccessful upgrade, 274–275
backing up an existing /usr/include directory, 266
Berkeley DB NSS (Name Service Switch) module, 253
Berkeley Internet Name Daemon (BIND) 9, 253
Berkeley Standard Distribution (BSD), 248
build process, procedural overview, 254–255
Burrows-Wheeler compression algorithm, 259
BusyBox utility, 266, 270
bzip2 utility, using, 259
compiling Glibc, 264
compiling into the glibc-build directory, 261
compiling with extensions or add-ons, 252
Comprehensive Perl Archive Network, 257
continuous enhancement and improvement of, 249
default installation directories, 262
DES cryptographic functions, 254
disk space required for building and installing, 247
--enable-add-ons command-line option, 253, 263
examining and setting the symbolic links, 272
executing multiple compilations on a multiprocessor system, 264
executing the Glibc gnu_get_libc_version() function, 251

executing the main Glibc shared library, 252
finding documentation on Glibc, 277
functions of, 247
GCC 4 or later, 256
Glibc configure script, --disable-sanity-checks command-line option, 262
Glibc configure script, --help command-line option, 263
Glibc configure script, --prefix command-line option, 257, 262–263, 268
Glibc test suite and Linux kernels version 2.6.16 or later, 249
Glibc version 2.4, supported Linux kernel configurations, 278
Glibc version 2.4, upgrading to, 278
Glibc Web sites and mailing lists, 277
glibc_version.c program, code example, 251
glibc-compat add-on, 254
glibc-crypt add-on, 254
glibc-linuxthreads add-on, 253
GNU awk 3.0, 256
GNU binutils 2.13 or later, 256
GNU make 3.79 or later, 256
GNU sed 3.02 or later, 256
GNU tar, downloading and using, 259
GNU Texinfo 3.12f or later, 257
gzip utility, downloading and using, 258–259
handling segmentation faults or errors, 271
i18n support, 250
identifying the currently used Glibc version, 251
INSTALL text file, 256, 263
installing an alternate Glibc, 268
installing Glibc as the primary C library, 266
ISO C standard, 248
kernel optimization for Linux systems, 264
Knoppix Linux distribution, 270
LANGUAGE environment variable, 255
LC_ALL environment variable, 255
Lempel-Ziv coding, 259
libidn library, 253, 260–261
libio (Glibc IO library), 254
libthreads module, 253
LinuxThreads add-on, 278
list of primary library links, 271
list of symbolic links to associated Glibc libraries, 271
listing the name of the Linux load library, 251
listing the name of the primary Glibc library, 251
localedata add-on, 254
ls and ln utilities, statically linked versions of, 270
major version changes of, 249
major, minor and release version numbers explained, 250
make check command, 265
make dvi command, 277
make install command, 266
make -v command, 256
making a rescue disk, 254, 269
Name Service Switch (NSS), 258
Native POSIX Threading Library (NPTL), 253, 278
nss-db add-on, 253
nss-lwres module, 253
obtaining Glibc version details, 252
Perl 5 or later, 257
platforms supporting Glibc version 2.3.6, 249–250
ports add-on, 279
POSIX awk, 256
POSIX, 247–248
precautions before installing Glibc, 265

Find it faster at http://superindex.apress.com

primary Glibc download site, 254
printing a copy of the system's /etc/fstab file, 266
problems using multiple versions of Glibc, 276
problems when running an old version of gawk, 265
RamFloppy rescue disk, 269
recommended add-on packages, 253
recommended development utilities for building Glibc, 256–257
resolving problems with symbolic links, code example, 267–268
resolving upgrade problems using a rescue disk, 273
resolving upgrade problems using the BusyBox utility, 271
reverting to a previous Glibc, 265–266
RIP rescue disk, 269
shutting a system down to single-user mode, 255
source code, building from, 249
source code, configuring for compilation, 261–263
source code, downloading and installing, 258
submitting Glibc problem reports, 278
SunOS, 248
SYSV Unix, 248
tar (tape archiver) format, 258
testing applications with alternate versions of Glibc, 268
testing the build, 265
troubleshooting Glibc installation problems, 270
understanding potential upgrade problems, 250
Unix variants and the C programming language, 247
updating GNU utilities, 257
upgrading to Glibc version 2.3.6, 249
upgrading to Glibc version 2.4, 249
using the -ansi option, 248
using the which command, 257
--version command-line option, 257
X/Open Portability Guide (XPG), 248
X/Open System Interface (XSI), 248
X/Open Unix, 248
GNU C
alias attribute, 22
aligned attribute, 25–26
alloca(), 17, 27
always_inline attribute, 22
ARM #pragmas, 29
__attribute__ keyword, 21, 24
__builtin_apply_args(), 14
__builtin_return(), 14
case ranges, 21
cdecl attribute, 22
.common section, 26
const attribute, 22
constructing function calls, 14–15
Darwin #pragmas, 29
declaring function attributes, 21
declaring section names, code example, 26
deprecated attribute, 22, 25
designated initializers, 19–20
dllexport and dllimport attributes, 23
fastcall attribute, 23
flatten attribute, 23
flexible array members, 16
__FUNCTION__, 28
function names as strings, 28–29
inline functions, 27
label declaration, code example, 11
labels as values, 12
lexical scoping, 13

__LINE__, 28
list of features, 10–11
local label, definition of, 11
locally declared labels, 11–12
malloc attribute, 23
mixed declarations and code, 21
mode attribute, 25
nested functions, 13–14
nocommon attribute, 26
noinline attribute, 22
nonconstant initializers, 19
nonnull attribute, 23
noreturn attribute, 23
packed attribute, 26
parameter forward declaration, 18
pointer arithmetic, 19
#pragmas, GCC support of, 29
__PRETTY_FUNCTION__, 28
pure attribute, 24
regparm attribute, 24
section attribute, 26
sequence point, definition of, 18
Solaris #pragmas, 29
stdcall attribute, 24
subscripting non-lvalue arrays, 18–19
transparent_union attribute, 27
Tru64 #pragmas, 30
typeof keyword, 15
unused attribute, 24, 27
used attribute, 24
using computed goto statements, 12
variable attributes, 25
variable-length automatic arrays, 17–18
variadic macros, 18
warn_unused_result attribute, 24
-Winline command-line option, 27
zero-length arrays, 15–17
GNU Coding Standards, 151, 153, 168
GNU Compiler Collection (GCC)
accessing Usenet news, 215
advantages of Usenet newsgroups, 216
COBOL for GCC mailing lists, 223
code coverage, 120
code profiling, 120, 133
Compilers Resources Page, 224
CVS (Concurrent Versions System) repository, 221–222
distinguishing between a help request and a problem report, 218
ELinks browser, 223
GCC for Palm OS mailing lists, 223
GCC Web site, 223
GCC-related mailing lists at gcc.gnu.org, 219
gcov (GNU Coverage application), 120, 133
Gnatsweb, 221
GNU GNATS database, 221
GNU Press, 225
gnu.g++.bug newsgroup, 218
gnu.g++.help newsgroup, 218
gnu.gcc.announce newsgroup, 217
gnu.gcc.bug newsgroup, 217
gnu.gcc.help newsgroup, 218
gprof (GNU profiler), 120, 133
information on alternate C libraries, 225
information on building cross-compilers, 224
Internet Relay Chat (IRC) clients and channels, 224
Links browser, 223
list of open-source newsreaders, 216–217
list of the primary GCC-related newsgroups, 217
moderated and unmoderated newsgroups, 216

Find it faster at http://superindex.apress.com

netiquette for GCC-related mailing lists, 222
Network News Transport Protocol (NNTP), 216
newsgroups, news servers and newsreaders, 215
open news servers, 216
publications on GCC and related topics, 225–226
read/write GCC mailing lists, 220–221
read-only GCC mailing lists, 221–222
sites for Linux cross-compiler information, 224
SourceForge.net, 223
Unix to Unix Copy Protocol (UUCP), 216
Usenet resources for GCC, 215
GNU Compiler Collection (GCC) 4.x optimization
aligning data to natural memory size boundaries, 110
automatic optimizations, 113
avoiding frame pointers, 108
code size, definition of, 111
conditional constant propagation (CCP), 107
dead code elimination (DCE), 108
dead store elimination (DSE), 108
disabling a single optimization option, 107
dominator tree, definition of, 108
eliminating multiple redundant register loads, 111
full redundancy elimination (FRE), 108
GENERIC representation, 106
GIMPLE representation, 106
instruction scheduler enhancements, 108
level 1 optimizations, 106–108
level 2 optimizations, 109–111
level 3 optimizations, 112
loop optimizations, 108
manual optimization flags, table of, 112–113
modifying an optimization through flag options, 107
-O and -O1 optimization switches, 106
-O and -O1, table of optimization options, 107–108
-O optimization switch, 106
-O2 and -Os, comparison of, 112
-O2 optimization switch, 107
-O2, table of optimization options, 109–110
-O3 optimization switch, 107
-O3, list of optimization options, 112
-OO optimization switch, 106
optimization to reduce code size and enhance speed, 108
optimizing floating-point operations, 113
-Os optimization switch, 106–107
-Os, features of, 111
-Os, list of disabled optimization flags, 111
partial redundancy elimination (PRE), 108
peephole optimizations, 111
processor-specific optimizations, 113
protection against buffer and stack overflows, 106
sibling call, 111
specifying multiple -O options, 112
static single assignment (SSA), 106
tail recursive call, 111
temporary expression replacement (TER), 108
Tree SSA, benefits of, 106
vectorization, 106
GNU Compiler Collection (GCC) optimization
benefits of converting code into intermediate representations, 105
Register Transfer Language (RTL), uses of, 105

GNU FORTRAN 77. *See* g77 compiler
GNU General Public License (GPL), 283
GNU Info
  @-commands, 492
  case sensitivity of searches, 499
  commands for working with multiple windows, 503
  comparison to the Unix manual page, 491
  components of an Info screen, 494
  Ctrl-x and Ctrl-x n, explanations of, 494
  definition of, 491
  documentation for, 495
  echo area (echo line), 495
  Emacs key bindings, 494
  executing from the command line, 501
  helpful tricks for using, 502
  hypertext capabilities of, 491, 499
  info command, 492, 498
  info command, table of command-line options, 501–502
  interface of, 491
  jumping to nodes, 497
  keyboard navigation commands, 493
  makeinfo command, 492
  menu reference, 496, 500
  Meta-x, explanation of, 494
  mode line, 495
  M-x print-node command, 500
  node (topic), 495
  node structure of, 497
  note reference, 500
  pressing Ctrl-g to cancel operations, 499
  pressing Ctrl-h for the help screen, 494
  pressing the l key to close the help window, 494
  pressing the l key to return to the last visited node, 497, 500
  pressing the Next (Page Down) and Previous (Page Up) keys, 496
  pressing the spacebar to scroll windows, 496
  printing a complete Info file, 500
  printing an individual node, 500
  repeating previous searches, 499
  rules for navigating Info files, 498
  search commands, 498
  selecting a node, 496
  selecting an xref with the cursor, 499
  starting and exiting, 492
  --subnodes option, 502
  Texinfo format text files, 492
  tutorial on, 494
  using command multipliers, 502
  using enscript to convert Info text to PostScript, 502
  using the cursor keys to scroll lines of text, 496
  using the Esc key instead of Alt, 494
  window (view area), 495
  xref (cross-reference), 499
GNU Library Tool. *See* Libtool
gprof
  addr2line utility, 134, 148
  annotated source code listing, 133
  binutils package, compiling and installing, 134–135
  binutils package, included utilities, 134
  call graph, 133, 144
  command-line options, table of, 137, 139, 140
  common profiling errors, 149
  compiling source code for the single source module, 146–147
  compiling the sample application using the -pg option, 140

Find it faster at http://superindex.apress.com

__cyg_profile_func_enter(), use of, 148
default output from a gprof test run, 141–143
displaying annotated source code for application functions, 144
flat profile, 133, 144
forms of profiling output, 133
function index, 144
-g option, 135
generating a single source module for the sample application, 146
gmon.out file, 135–136
inserting user-defined profiling code in functions, 148
mapping addresses to function names, 148–149
output from the compiled sample application, 150
-pg option, 135
requirements for compiling code for profile analysis, 135
running gprof with the -A option, 145
running gprof with the -b option, 144
setting the GPROF_PATH environment variable, 140
source code for the sample application, 149–150
support for the BSD Unix prof application, 133
symbol specifications, syntax of, 136
using the -finstrument-functions option, 135, 148–150
writing custom profiling routines, 134

H

H8/300 options
H8/300-based microprocessors, 428
-malign-300, 429
-mh, 429
-mint32, 429
-mn, 429
-mrelax, 429
-ms, 429
-ms2600, 429
HP/PA (PA/RISC) options
DCE (Distributed Computing Environment) thread library, 431
Hewlett-Packard Precision Architecture, 429
-march=architecture-type, 429
-mbig-switch, 429
-mdisable-fpregs, 429
-mdisable-indexing, 429
-mfast-indirect-calls, 429
-mgas, 430
-mgnu-ld, 430
-mhp-ld, 430
-mjump-in-delay, 430
-mlinker-opt, 430
-mlong-calls, 430
-mlong-load-store, 430
-mno-space-regs, 430
-mpa-risc-1-0, 430
-mpa-risc-1-1, 430
-mpa-risc-2-0, 430
-mportable-runtime, 430
-mschedule=CPU-type, 430
-msio, 430
-msoft-float, 430
-munix=UNIX-STD, 431
-mwsio, 431
PA-RISC microprocessor architecture, 429
-static, 431
-threads, 431
VLSI Technology Operation, 429

i386 and AMD x86-64 options
3DNow extensions, 432
AMD64 64-bit processors, 431
i386 processors, 431
-m128bit-long-double, 431
-m32, 431
-m386, 431
-m3dnow, 432
-m486, 432
-m64, 432
-m96bit-long-double, 432
-maccumulate-outgoing-args, 432
-malign-double, 432
-march=CPU-type, 432
-masm=dialect, 432
-mcmodel=model, 432
-mcpu=CPU-type, 432
-mfpmath=unit, 432
-mieee-fp, 433
-minline-all-stringops, 433
-mlarge-data-threshold=number, 433
-mmmx, 433
-mno-3dnow, 433
-mno-align-double, 433
-mno-align-stringops, 433
-mno-fancy-math-387, 433
-mno-fp-ret-in-387, 434
-mno-ieee-fp, 434
-mno-mmx, 434
-mno-push-args, 434
-mno-red-zone, 434
-mno-sse, 434
-mno-sse2, 434
-mno-svr3-shlib, 434
-mno-tls-direct-seg-refs, 434
-momit-leaf-frame-pointer, 434
-mpentium, 434
-mpentiumpro, 434
-mpreferred-stack-boundary=num, 434
-mpush-args, 435
-mregparm=num, 435
-mrtd, 435
-msoft-float, 435
-msse, 436
-msse2, 436
-msselibm, 436
-msseregparm, 436
-msvr3-shlib, 436
-mthreads, 436
-mtls-direct-seg-refs, 436
-mtune=CPU-type, 436
SSE extensions, 436
SSE2 extensions, 434
System V Release 3 (SVR3) systems, 434, 436
IA-64 options
AIX 5 systems, 438
DWARF2 line number debugging, 438
HP-UX systems, 438
Intel 64-bit processors, 437
-mauto-pic, 437
-mbig-endian, 438
-mb-step, 437
-mconstant-gp, 438
-mdwarf2-asm, 438
-mearly-stop-bits, 438
-mfixed-range=register-range, 438
-mgnu-as, 438
-mgnu-ld, 438
-milp32, 438
-milp64, 438
-minline-float-divide-max-throughput, 438
-minline-float-divide-min-latency, 438
-minline-int-divide-max-throughput, 438

Find it faster at http://superindex.apress.com

-minline-int-divide-min-latency, 438
-minline-sqrt-max-throughput, 438
-minline-sqrt-min-latency, 438
-mlittle-endian, 438
-mno-dwarf2-asm, 438
-mno-early-stop-bits, 438
-mno-gnu-as, 438
-mno-gnu-ld, 439
-mno-inline-float-divide, 439
-mno-inline-int-divide, 439
-mno-pic, 439
-mno-register-names, 439
-mno-sched-ar-data-spec, 439
-mno-sched-ar-in-data-spec, 439
-mno-sched-br-data-spec, 439
-mno-sched-br-in-data-spec, 439
-mno-sched-contol-ldc, 439
-mno-sched-control-spec, 439
-mno-sched-count-spec-in-critical-path, 439
-mno-sched-in-control-spec, 439
-mno-sched-ldc, 439
-mno-sched-prefer-non-control-spec-insns, 439
-mno-sched-spec-verbose, 439
-mno-sdata, 439
-mno-volatile-asm-stop, 439
-mregister-names, 440
-msched-ar-data-spec, 440
-msched-ar-in-data-spec, 440
-msched-br-data-spec, 440
-msched-br-in-data-spec, 440
-msched-contol-ldc, 440
-msched-control-spec, 440
-msched-count-spec-in-critical-path, 440
-msched-in-control-spec, 440
-msched-ldc, 440
-msched-prefer-non-control-spec-insns, 440
-msched-prefer-non-data-spec-insns, 439–440
-msched-spec-verbose, 440
-msdata, 440
-mt, 439
-mtls-size=tls-tls, 440
-mtune=CPU-type, 440
-mvolatile-asm-stop, 441
Native POSIX Threading Library (NPTL), 439, 441
-pthread, 441
register range, 438
inlining loops, defined, 119
Intel 960 options
iC960 assembler, 441
-masm-compat, 441
-mcode-align, 441
-mcomplex-addr, 441
-mcpu=CPU-type, 441
-mic2.0-compat, 441
-mic3.0-compat, 442
-mic-compat, 441
-mintel-asm, 441
-mleaf-procedures, 442
-mlong-double-64, 442
-mno-code-align, 442
-mno-complex-addr, 442
-mno-leaf-procedures, 442
-mno-strict-align, 442
-mno-tail-call, 442
-mnumerics, 442
-mold-align, 442
-msoft-float, 442
-mstrict-align, 442
-mtail-call, 442

J

jar files
- creating a Manifest.txt file, 91–92
- creating a shared library from a jar file, 92
- creating and using, 90–92
- definition of, 90
- jar utility and Unix/Linux tar application, 90

Java Virtual Machines (JVMs)
- BEA's WebLogic JRockit, 79
- Blackdown's Java Platform 2 for Linux, 79
- HotSpot compilers, 79
- IBM's Java 2 Runtime Environment, 79
- Just-in-Time (JIT) compilers, 79
- Kaffe, 79
- SableVM, 79
- Sun's HotSpot Client and Server JVMs, 79

K

Kegel, Dan, 300

klibc
- building, 287
- compiling an application statically with klibc, code example, 288
- configuring the Linux kernel source manually, 287
- cpio-formatted initramfs image, 286
- cross-compiling, 288
- embedded systems and, 286
- executing the make test command, 288
- information resources on, 286
- initramfs-style initial RAM disk, 286
- initrd ext2 filesystem image, 286
- licensing under the GNU General Public License (GPL), 286
- Linux systems and, 286
- obtaining and retrieving, 286
- using with gcc, 288

L

Ladd, Scott, 114

libraries. *See* code libraries

Libtool
- autoconf and, 181, 185, 194
- automake and, 181, 185, 194
- clean mode, 187
- code libraries, definition of, 177
- command-line modes and options, 186
- compile mode, 187
- configure.ac file with Libtool integration, code example, 192
- configure.ac file, code example, 191
- conventions in library extensions, 181
- converting code modules into a library, 193
- execute mode, 187–188
- finish mode, 188
- home page and downloadable archive, 182
- install mode, code example, 188–189
- installed files and directories, 184–185
- installing and building, 182–184
- .la files, 181
- LD_LIBRARY_PATH environment variable, 187–188
- link mode, command-line options, 189–190
- linking applications against multiple shared libraries, 181
- .lo (library object) files, 181
- Makefile.am file with Libtool integration, code example, 192
- manually creating Makefiles, code example, 190–191
- --mode command-line option, 186
- obtaining further information on, 195
- operation of, 181
- position-independent code (PIC) objects, 187

Find it faster at http://superindex.apress.com

purpose of, 177
recommended version, 182
Red Hat Package Manager (RPM), 183
troubleshooting common errors, 194–195
using from the command line, 185
Linux Standard Base, compression utilities and, 260

M

M32C options
-mcpu=name, 443
-memregs=number, 443
-msim, 443
M32R options
-G num, 443
-m32r, 443
-m32r2, 443
-m32rx, 443
-malign-loops, 443
-mbranch-cost=number, 443
-mcode-model=large, 444
-mcode-model=medium, 444
-mcode-model=small, 444
-mdebug, 444
-mflush-func=name, 444
-mflush-trap=number, 444
-missue-rate=number, 444
-mno-align-loops, 444
-mno-flush-func, 444
-mno-flush-trap, 444
-msdata=none, 444
-msdata=sdata, 444
-msdata=use, 444
Renesas M32R processor family, 443
M680x0 options
-m5200, 445
-m68000, 445
-m68020, 445
-m68020-40, 445
-m68020-60, 445
-m68030, 445
-m68040, 445
-m68060, 445
-m68881, 445
-malign-int, 445
-mbitfield, 446
-mc68000, 445
-mc68020, 445
-mcfv4e, 446
-mcpu32, 446
-mfpa, 446
-mid-shared-library, 446
-mno-align-int, 446
-mnobitfield, 446
-mno-id-shared-library, 446
-mno-sep-data, 446
-mno-strict-align, 446
Motorola 68000 family of processors, 445
-mpcrel, 446
-mrtd, 446
-msep-data, 447
-mshared-library-id=n, 447
-mshort, 447
-msoft-float, 447
-mstrict-align, 447
M68HC1x options
-m6811, 447
-m6812, 447
-m68hc11, 447
-m68hc12, 447
-m68hcs12, 447
-m68S12, 447
-mauto-incdec, 447
-minmax, 447

-mlong-calls, 447
-mno-long-calls, 447
-mshort, 448
-msoft-reg-count=count, 448
-no-minmax, 447
M88K options
88open Object Compatibility Standard, 449
Data General AViiON workstation, 448
-m88000, 448
-m88100, 448
-m88110, 448
-mbig-pic, 448
-mcheck-zero-division, 448
-mhandle-large-shift, 448
-midentify-revision, 448
-mno-check-zero-division, 448
-mno-ocs-debug-info, 449
-mno-ocs-frame-position, 449
-mno-optimize-arg-area, 449
-mno-serialize-volatile, 449
-mno-underscores, 449
-mocs-debug-info, 449
-mocs-frame-position, 449
-moptimize-arg-area, 449
Motorola 88000 family of RISC processors, 448
-mserialize-volatile, 449
-mshort-data-num, 449
-msvr3, 450
-msvr4, 450
-mtrap-large-shift, 450
-muse-div-instruction, 450
-mversion-03.00, 450
-mwarn-passed-structs, 450
Mach operating system, 423
Macintosh OS X operating system, 30, 74, 423
MacKenzie, David, 152–153
MCore options
-m210, 450
-m340, 450
-m4byte-functions, 450
-mbig-endian, 450
-mcallgraph-data, 450
-mdiv, 450
-mhardlit, 451
-mlittle-endian, 451
-mno-4byte-functions, 451
-mno-callgraph-data, 451
-mno-div, 451
-mno-hardlit, 451
-mno-relax-immediate, 451
-mno-slow-bytes, 451
-mno-wide-bitfields, 451
Motorola MCore processor family, 450
-mrelax-immediate, 451
-mslow-bytes, 451
-mwide-bitfields, 451
MIPS options
-EB, 451
ECOFF binary output format, 453
-EL, 451
ELF binary output format, 453
-G num, 451
-m4650, 452
-mabi=32, 452
-mabi=64, 452
-mabi=eabi, 452
-mabi=n32, 452
-mabi=o64, 452
-mabicalls, 452
-march=arch, 452
-mbranch-likely, 452
-mcheck-zero-division, 453
-mdivide-breaks, 453

Find it faster at http://superindex.apress.com

-mdivide-traps, 453
-mdouble-float, 453
-mdsp, 453
-membedded-data, 453
-membedded-pic, 453
-mentry, 453
-mexplicit-relocs, 453
-mfix-r4000, 453
-mfix-r4400, 453
-mfix-sb1, 453
-mfix-vr4120, 453
-mfix-vr4130, 453
-mflush-func=func, 453
-mfp32, 454
-mfp64, 454
-mfp-exceptions, 454
-mfused-madd, 454
-mgas, 454
-mgp32, 454
-mgp64, 454
-mgpopt, 454
-mhalf-pic, 454
-mhard-float, 454
-mint64, 454
MIPS ISA (Instruction Set Architecture), 452
-mips1, 454
-mips16, 454
-mips2, 454
-mips3, 454
-mips3d, 454
-mips4, 455
-mlong32, 455
-mlong64, 455
-mlong-calls, 455
-mmad, 455
-mmemcpy, 455
-mno-abicalls, 455
-mno-branch-likely, 455
-mno-check-zero-division, 455
-mno-dsp, 455
-mno-embedded-data, 456
-mno-embedded-pic, 456
-mno-explicit-relocs, 455
-mno-flush-func, 455
-mno-fp-exceptions, 455
-mno-fused-madd, 455–456
-mno-gpopt, 456
-mno-half-pic, 456
-mno-long-calls, 456
-mno-mad, 456
-mno-memcpy, 456
-mno-mips16, 456
-mno-mips3d, 455
-mno-mips-tfile, 456
-mno-rnames, 456
-mno-shared, 456
-mno-split-addresses, 456
-mno-stats, 456
-mno-sym32, 456
-mno-uninit-const-in-rodata, 457
-mno-vr4130-align, 457
-mno-xgot, 457
-mshared, 457
-msingle-float, 457
-msoft-float, 457
-msplit-addresses, 457
-mstats, 457
-msym32, 457
-mtune=arch, 457
-muninit-const-in-rodata, 458
-mvr4130-align, 458
-mxgot, 458
-nocpp, 458
-no-crt0, 458

MMIX options
-mabi=gnu, 458
-mabi=mmixware, 458
-mbase-addresses, 458
-mbranch-predict, 458
-melf, 458
-mepsilon, 458
-mknuthdiv, 458
-mlibfuncs, 458
-mno-base-addresses, 458
-mno-branch-predict, 459
-mno-epsilon, 459
-mno-knuthdiv, 459
-mno-libfuncs, 459
-mno-single-exit, 459
-mno-toplevel-symbols, 459
-mno-zero-extend, 459
-msingle-exit, 459
-mtoplevel-symbols, 459
-mzero-extend, 459
PREFIX assembly directive, 459
MN10200 options
MN10200 16-bit single-chip microcontrollers, 459
-mrelax, 459
MN10300 options
-mam33, 460
-mmult-bug, 460
-mno-am33, 460
-mno-crt0, 460
-mno-mult-bug, 460
-mno-return-pointer-on-d0, 460
-mrelax, 460
-mreturn-pointer-on-d0, 460
MT options
-march=cpu-type, 460
-mbacc, 460
-mno-bacc, 460
-mno-crt0, 460
Morpho Technologies MS1 and MS2 processors, 460
-msim, 460

N

Native POSIX Threading Library (NPTL), 253, 278, 439, 441, 482
Newlib
embedded systems, 289
frustrations in using, 289
integration with GCC, 289
licensing of, 289
Newlib FTP directory, 290
obtaining and retrieving, 289
using with cross-compilers, 289
-with-newlib option, 289
NeXTSTEP operating system, 30
NS32K options
-m32032, 461
-m32081, 461
-m32332, 461
-m32381, 461
-m32532, 461
-mbitfield, 461
-mhimem, 461
-mmulti-add, 461
-mnobitfield, 461
-mnohimem, 461
-mnomulti-add, 461
-mnoregparam, 461
-mnosb, 461
-mregparam, 462
-mrtd, 462
-msb, 462
-msoft-float, 462

Find it faster at http://superindex.apress.com

O

Objective-C
- Boehm-Demers-Weiser conservative garbage collector, 37
- @catch block, 38
- class_ivar_set_gcinvisible() runtime function, 37
- constant string objects, defining and declaring, 36–37
- declaring weak pointer references, 37
- development of, 30
- errors from missing Objective-C runtime library, 33
- executing the +load class load mechanism, 37
- @finally block, 38
- -fconstant-string-class option, 36
- garbage collection, 37
- gcc C compiler and, 30
- GCC compilation options, 33–35
- Hello, World program, code example, 32
- identifying library dependencies, 30–31
- information resources on, 31
- +initialize mechanism, 37
- ldd application, 30
- list of supported @keyword statements, 32–33
- list of type encodings, 39–40
- .m extension, 32
- Mac OS X systems, 30
- NeXTSTEP operating system and, 30
- NXConstantString class, 36
- runtime libraries, 30
- Smalltalk-80 language and, 30
- specifying the –lobjc option when linking, 34
- structured error handling, code example, 37–38
- @synchronized block, 38
- @throw statement, 38
- using synchronization blocks for thread-safe execution, 38

OpenMP API Specification, 371

P

PDP-11 options
- -m10, 462
- -m40, 462
- -m45, 462
- -mabshi, 462
- -mac0, 462
- -mbcopy, 462
- -mbcopy-builtin, 462
- -mbranch-cheap, 462
- -mbranch-expensive, 462
- -mdec-asm, 463
- -mfloat32, 463
- -mfloat64, 463
- -mfpu, 463
- -mint16, 463
- -mint32, 463
- -mno-abshi, 463
- -mno-ac0, 463
- -mno-float32, 463
- -mno-float64, 463
- -mno-int16, 463
- -mno-int32, 463
- -mno-split, 463
- -msoft-float, 463
- -msplit, 463
- -munix-asm, 463
- PDP-11 minicomputers, 462
- split Instruction and Data spaces, 463

Perl interpreter
Comprehensive Perl Archive Network, 257
downloading source code for, 159
Perl 5 or later, 257
Perl scripts and auxiliary utilities, table of, 158
Pluggable Authentication Modules (PAM), 180
POSIX, 247–248
Native POSIX Threading Library (NPTL), 253, 278, 439, 441, 482
POSIX awk, 256
PowerPC (PPC) options
AIX systems, 466
AltiVec ABI extensions, 464
Apple Macintosh computer systems, 464
Apple-IBM-Motorola alliance, 463
-G num, 464
IBM RS/6000 workstation, 463
IBM RS64 processor family, 464
IBM XL compilers, 470, 473
-m32, 464
-m64, 464
-mabi=abi-type, 464
-mads, 464
-maix32, 464
-maix64, 464
-maix-struct-return, 464
-malign-natural, 465
-malign-power, 465
-maltivec, 465
-mbig, 465
-mbig-endian, 465
-mbit-align, 465
-mbss-plt, 465
-mcall-aix, 465
-mcall-gnu, 465
-mcall-linux, 465
-mcall-netbsd, 465
-mcall-solaris, 465
-mcall-sysv, 465
-mcall-sysv-eabi, 465
-mcall-sysv-noeabi, 465
-mcpu=cpu-type, 466
-mdlmzb, 466
-mdynamic-no-pic, 466
-meabi, 466
-memb, 466
-mfloat-gprs, 467
-mfprnd, 467
-mfull-toc, 467
-mfused-madd, 467
-mhard-float, 467
-minsert-sched-nops=scheme, 467
-misel, 467
-mlittle, 467
-mlittle-endian, 467
--mlongcall, 467
-mmfcrf, 467
-mminimal-toc, 467
-mmulhw, 468
-mmultiple, 468
-mmvme, 468
-mnew-mnemonics, 468
-mno-altivec, 468
-mno-bit-align, 468
-mno-dlmzb, 468
-mno-eabi, 468
-mno-fp-in-toc, 468
-mno-fprnd, 468
-mno-fused-madd, 468
-mno-isel, 468
-mno-longcalls, 468
-mno-mfcrf, 469
-mno-mulhw, 469

Find it faster at http://superindex.apress.com

-mno-multiple, 469
-mno-popcntb, 469
-mno-power, 469
-mno-power2, 469
-mno-powerpc, 469
-mno-powerpc64, 469
-mno-powerpc-gfxopt, 469
-mno-powerpc-gpopt, 469
-mno-prototype, 469
-mno-regnames, 469
-mno-relocatable, 470
-mno-relocatable-lib, 470
-mno-secure-plt, 470
-mno-spe, 470
-mno-strict-align, 470
-mno-string, 470
-mno-sum-in-toc, 470
-mno-swdiv, 470
-mno-toc, 470
-mno-update, 470
-mno-vrsave, 470
-mno-xl-compat, 470
-mold-mnemonics, 470
-mpe, 471
-mpopcntb, 471
-mpower, 471
-mpower2, 471
-mpowerpc, 471
-mpowerpc64, 471
-mpowerpc-gfxopt, 471
-mpowerpc-gpopt, 471
-mprioritize-restricted-insns=priority, 471
-mprototype, 472
-mregnames, 472
-mrelocatable, 472
-mrelocatable-lib, 472
-msched-costly-dep=dependence-type, 472
-msdata=abi, 472
-msdata-data, 472
-msecure-plt, 472
-msim, 472
-msoft-float, 472
-mspe, 473
-mstrict-align, 473
-mstring, 473
-msvr4-struct-return, 473
-mswdiv, 473
-mtoc, 473
-mtune=cpu-type, 473
-mupdate, 473
-mvrsave, 473
-mvxworks, 473
-mwindiss, 473
-mxl-compat, 473
-myellowknife, 473
Parallel Environment (PE), 471
Performance Optimization With Enhanced RISC (POWER), 463
-pthread, 474
SPE SIMD instructions, 464
printf()
debugging with, 119
problems masked by, 119

R

Red Hat, 152, 159
Red Hat Package Manager (RPM), 183
Register Transfer Language (RTL)
advantages and disadvantages of using, 105
RT options
Academic Operating System (AOS), 474
AIX operating system, 474
Mach operating system, 474
-mcall-lib-mul, 474

-mfp-arg-in-fpregs, 474
-mfp-arg-in-gregs, 474
-mfull-fp-blocks, 474
-mhc-struct-return, 474
-min-line-mul, 474
-mminimum-fp-blocks, 474
-mnohc-struct-return, 474
-mpcc-struct-return, 474

S

S/390 and zSeries options
ESA/90 ABI, 475
-m31, 475
-m64, 475
-march=CPU-type, 475
-mbackchain, 475
-mdebug, 475
-mesa, 475
-mfused-madd, 475
-mguard-size=GUARD-SIZE, 475
-mhard-float, 475
-mlong-double-128, 475
-mlong-double-64, 475
-mmvcle, 475
-mno-backchain, 475
-mno-debug, 475
-mno-fused-madd, 476
-mno-mvcle, 476
-mno-packed-stack, 476
-mno-small-exec, 476
-mno-tpf-trace, 476
-mpacked-stack, 476
-msmall-exec, 476
-msoft-float, 476
-mstack-size=stack-size, 476
-mtpf-trace, 476
-mvcle, 476
-mwarn-dynamicstack, 476
-mwarn-framesize=FRAMESIZE, 476
-mzarch, 476
SH options
-m1, 477
-m2, 477
-m2e, 477
-m3, 477
-m3e, 477
-m4, 477
-m4a, 477
-m4al, 477
-m4a-nofpu, 477
-m4a-single, 477
-m4a-single-only, 477
-m4-nofpu, 477
-m4-single, 477
-m4-single-only, 477
-madjust-unroll, 477
-mb, 477
-mbigtable, 477
-mdalign, 477
-mdiv=strategy, 477
-mdivsi3_libfunc=name, 478
-mfmovd, 478
-mgettrcost=number, 478
-mhitachi, 478
-mieee, 478
-mindexed-addressing, 478
-minvalid-symbols, 478
-misize, 478
-ml, 478
-mnomacsave, 478
-mno-pt-fixed, 478
-mno-renasas, 478
-mpadstruct, 478
-mprefergot, 478

Find it faster at http://superindex.apress.com

-mpt-fixed, 478
-mrelax, 478
-mrenasas, 478
-mspace, 478
-multcost=number, 479
-musermode, 479
-no-minvalid-symbols, 478
SHmedia SIMD instruction set, 477
SuperH (SH) processors, 477
SourceForge.net, 223
SPARC options
-m32, 479
-m64, 479
-mapp-regs, 479
-mcmodel=code-model, 479
-mcmodel=embmedany, 479
-mcmodel=medany, 479
-mcmodel=medlow, 479
-mcmodel=medmid, 480
-mcpu=CPU-type, 480
-mcypress, 480
Medium/Anywhere code model, 479
Medium/Low code model, 479
Medium/Middle code model, 480
-mfaster-structs, 480
-mflat, 480
-mfpu, 480
-mhard-float, 480
-mhard-quad-float, 480
-mimpure-text, 481
-mlittle-endian, 481
-mno-app-regs, 481
-mno-faster-structs, 481
-mno-flat, 481
-mno-fpu, 481
-mno-stack-bias, 481
-mno-unaligned-doubles, 481
-mno-v8plus, 481
-mno-vis, 481
-msoft-float, 481
-msoft-quad-float, 482
-msparclite, 482
-mstack-bias, 482
-msupersparc, 482
-mtune=cpu-type, 482
-munaligned-doubles, 482
-mv8plus, 482
-mvis, 482
Native POSIX Threading Library (NPTL), 482
-pthread, 482
-pthreads, 482
SPARC processors, 479
Sun Microsystems, 479
-threads, 482
UltraSPARC family, 479
UltraSPARC VIS (Visual Instruction Set), 481
Subversion Source Code Control System (SCCS), 308
System V options
-G, 483
-Qn, 483
-Qy, 483
System V Release 4 (SVR4), 482
-Ym,dir, 483
-YP,dir, 483

T

Tcl (Tool Command Language), 229, 242
test coverage
basic block (statement) coverage, 121
decision (branch) coverage, 121
defined, 120
modified condition decision (expression) coverage, 121
path (predicate) coverage, 121

test suites
call graphs, 119
designing, 119, 121
difficulties in reproducing improbable errors, 121–122
fprintf() statement, 122
providing both statement and decision coverage, 122–123
TMS320C3x/C4x options
interrupt service routine (ISR), 483
-mbig, 483
-mbig-memory, 483
-mbk, 483
-mcpu=cpu-type, 483
-mdb, 483
-mdp-isr-reload, 483
-mfast-fix, 484
-mloop-unsigned, 484
-mmemparm, 484
-mmpyi, 484
-mno-bk, 484
-mno-db, 484
-mno-fast-fix, 484
-mno-loop-unsigned, 484
-mno-mpyi, 484
-mno-parallel-insns, 484
-mno-parallel-mpy, 484
-mno-rptb, 484
-mno-rpts, 484
-mparallel-insns, 484
-mparallel-mpy, 485
-mparanoid, 483
-mregparm, 485
-mrptb, 485
-mrpts=count, 485
-msmall, 485
-msmall-memory, 485
-mti, 485
troubleshooting GCC
adjusting the PATH environment variable, 200
AIX systems, 207
alias gcc command, 203
benefits of using the GNU C library (Glibc), 209
“Cannot execute binary file” errors, 203
checking GCC’s Info file, 206
checking online version-specific GCC documentation, 198
checking the build logs or output window for error messages, 202
compatibility problems when using third-party tools, 206–208
correct use of __STDC__, 211
cross-compiler problems, 203
/etc/ld.so.conf text file, 201
executing the file filename command, 203
executing the make –n command with a modified Makefile, 206
fixincludes script, 208, 212
fixproto script, 198
generating position-independent code (PIC), 207
incompatibilities between GNU C and K&R C, 210–211
issues in not using GNU make, 205
ldconfig command, 201
linking an application statically, 202
malloc function on Solaris systems, 208
malloc() replacements, 203
misfeatures, definition of, 198
mixing GNU and third-party tools, 204–206
moving GCC after installation, 204
multiple GCC installations on a single system, 200
mysterious warning and error messages, 209–210
“No such file or directory” errors, 202

Find it faster at http://superindex.apress.com

optimization problems, 208
-pedantic option (gcc compiler), 209
-pedantic-errors option (gcc compiler), 198, 209
potential dangers in using symbolic links, 204
--print-search-dirs option, 205
problems executing files, 203
problems executing GCC, 200
problems with include files or libraries, 208–209
problems with the shared library loader, 202
reading GCC-related newsgroups, 198
regenerating GCC's header files, 208
resolving build and installation problems, 212–213
resolving shared library problems, 201–202
restrictive file permissions or ACLs (access control lists), 200
running out of memory, 203
-traditional option (gcc compiler), 210–211
using an up-to-date version of GDB, 207
using the -### option, 199
using the BusyBox binary, 203
using verbose modes, 205
-Wall option, 209
warnings and errors differentiated, 209
whereis gcc command, 201
which gcc command, 201, 203
workarounds for fixincludes script and automounter problems, 198

U

uClibc
building, 292
buildroot project, 291
configuring, 292–293, 295–296
disabling shared library support, 294
executing the make menuconfig command, 292
home page of, 290
Library Installation Options dialog, 295
licensing under the LGPL, 290
obtaining and retrieving, 291
operation on standard and MMU-less processors, 290
selecting processor-specific configuration parameters, 292
specifying the target architecture and processor family, 292, 294
subversion SCCS (Source Code Control System), 291
support for binary compatibility, 295
support for command-line and terminal-oriented configuration, 291
Target Architecture dialog, 292
uClibc patches and the mainline Linux kernel, 291
use on embedded systems and rescue disks, 290
using with gcc, 296
Unix
AIX operating system, 474
AT&T's variants of, 151
autoconf program, 152
automake program, 153
Berkeley Standard Distribution (BSD), 151, 248, 423
bmake program, 153
C programming language and Unix variants, 247
compiling conditional code using #ifdef, 151–152
Cygnus Solutions, 152
differences between implementations, 151

Feldman, Stu, 153
gcc, 152
generating platform-specific Makefiles, 151
GNU make program, 153
imake program, 153
m4 macro processor, 152
make program, functions of, 153
Makefiles, explanation of, 153
metaconfig program, 152
resolving portability issues, 151
Unix to Unix Copy Protocol (UUCP), 216
variants of, 151
unrolling loops, defined, 119

V

V850 options
-mapp-regs, 485
-mbig-switch, 485
-mdisable-callt, 485
-mep, 485
-mlong-calls, 486
-mno-app-regs, 486
-mno-disable-callt, 486
-mno-ep, 486
-mno-long-calls, 486
-mno-prolog-function, 486
-mprolog-function, 486
-msda=n, 486
-mspace, 486
-mtda=n, 486
-mv850, 486
-mv850e, 486
-mv850e1, 486
-mzda=n, 486
NEC V850 32-bit RISC microcontrollers, 485
VAX FORTRAN, 75
VAX options
-mg, 487
-mgnu, 487
-munix, 487
Very Long Instruction Word (VLIW) processor, 425
Visualization of Compiler Graphs (VCG), 347
VMS operating system, 408

X

X/Open Unix, 248
Xstormy16 options
-msim, 487
Sanyo Xstormy16 processor, 487
Xtensa options
-mbig-endian, 487
-mbooleans, 487
-mconst16, 487
-mdensity, 487
-mfused-madd, 488
-mhard-float, 488
-mlittle-endian, 488
-mlongcalls, 488
-mmac16, 488
-mminmax, 488
-mmul16, 488
-mmul32, 488
-mno-booleans, 488
-mno-const16, 488
-mno-density, 488
-mno-fused-madd, 489
-mno-longcalls, 488
-mno-mac16, 489
-mno-minmax, 489
-mno-mul16, 489
-mno-mul32, 489
-mno-nsa, 489

Find it faster at http://superindex.apress.com

-mno-serialize-volatile, 489
-mno-sext, 489
-mno-target-align, 489
-mno-text-section-literals, 489
-mnsa, 489
-mserialize-volatile, 490
-msext, 490
-msoft-float, 490
-mtarget-align, 490
-mtext-section-literals, 490
normalization shift amount (NSA) instructions, 489
optional sign extend (SEXT) instruction, 489
system-on-a-chip (SoC) implementation, 487
Xtensa Processor Generator, 487